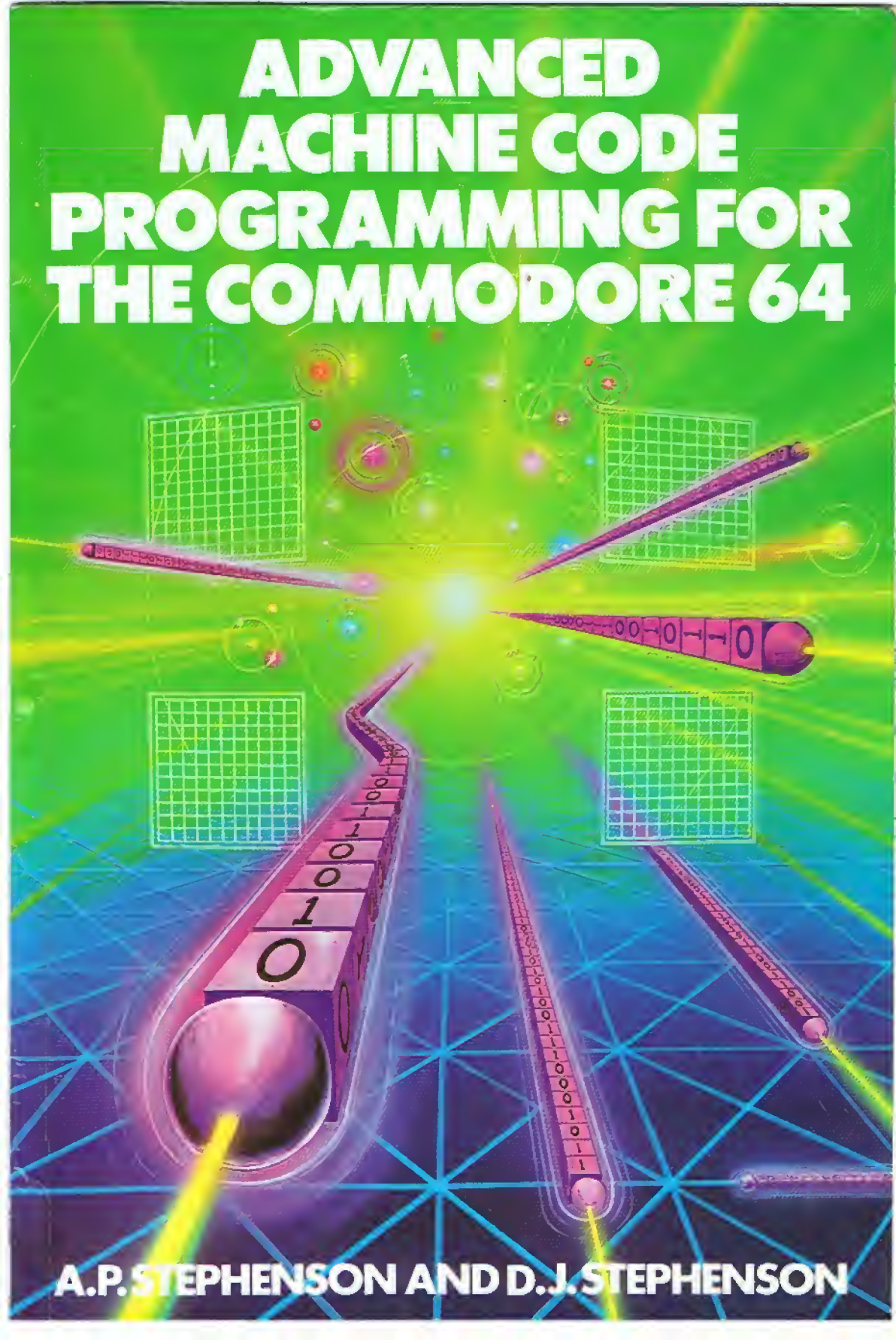
ADVANCED MACHINE CODE PROGRAMMING FOR THE COMMODORE 64



A.P. STEPHENSON AND D.J. STEPHENSON

Advanced Machine Code Programming for the Commodore 64

GRANADA
London Toronto Sydney New York

Granada Technical Books
Granada Publishing Ltd
8 Grafton Street, London W1X 3LA

First published in Great Britain by
Granada Publishing 1984

British Library Cataloguing in Publication Data

Stephenson, A. P.
Advanced machine code programming
for the Commodore 64.
1. Commodore 64 (Computer) - Programming
2. Machine codes (Electronic computers)
I. Title II. Stephenson, D. J.
001.64'25 QA76.8.C64

ISBN 0 246 12442-3

Typeset by V & M Graphics Ltd, Aylesbury, Bucks
Printed and bound in Great Britain by
Mackays of Chatham, Kent

Contents

Preface

This book is intended to complement Ian Sinclair's *Introducing Commodore 64 Machine Code*, also published by Granada. The word 'Advanced' appears in our title although this should in no way deter beginners providing they possess tenacity and sufficient dedication. It is assumed, however, that readers already have a little experience with BASIC.

Machine code has always had the reputation of being abstruse, difficult and prone to error. It is certainly more difficult than BASIC but, providing you invest in an assembler, not greatly so. If you intend to master machine code, an assembler is a necessity. If you try and muddle along with POKE codes there is a real danger that you will become frustrated and give up. Failure, in this case, would be due to the problems involved with entering and debugging, rather than difficulties intrinsic to the code. All programs in this book have been developed on a MIKRO 64 assembler, which is obtainable in plug-in cartridge form.

The microprocessor in the Commodore 64 is a 6510A which, for all practical purposes, is identical to the well-known 6502. Consequently, a good deal of the programming material in this book may be found useful to owners of other machines employing the 6502.

Although it is possible to use machine code without any hardware knowledge, it is certainly an advantage – and satisfying – to be aware of the various components and layout of the machine. In fact, it is not uncommon for beginners in machine code to develop an interest in the hardware side of the art, simply because studying machine code tends to generate such interest. Because of this, a certain amount of space has been devoted to aspects of hardware which might overlap software. A brief outline of TTL logic is given in the last chapter for those who like designing and/or interfacing their own circuitry for attachment to the user port or expansion bus.

A complete chapter is devoted to sort procedures and another to graphics because the speed advantage of machine code over BASIC is most evident in these two areas.

A. P. and D. J. Stephenson

Chapter One
Introduction

Virtues and faults of BASIC

Most users of personal computers either buy software ready-made or else they write their own programs in BASIC. This is natural. After all, BASIC was initially designed to be a pleasant and easy language to learn and use. Since it was first launched on 1st May 1964, there have been changes, some of which could be considered improvements, in both 'user-friendliness' and speed of execution. As far as possible, modern dialects of BASIC have reached the stage where the computer itself need hardly be considered. The language attempts to insulate the user not only from the harsh reality of the system software but also from the appalling complexity of the hardware.

Variations in BASIC dialects

There are one or two aspects of the language which can be irritating. First, BASIC, unlike the more traditional languages, is far from being standardised. Although broadly similar, there are significant variations between different versions of BASIC because software designers (software 'engineers' as they are now called) try to mould a particular version in a way which takes full advantage of features which are peculiar to a specific machine. Consequently, BASIC cannot be considered a standardised language since it is very unlikely that a program written for one brand of computer will run on another make without modifications to suit the change in the machine's 'personality'.

One reason for this is the inevitable result of the rapid progress in technology. In the era when BASIC was in its infancy, there was no chance of laser beam noises screeching away underneath the keyboard. Television sets were not used as screen displays then so there were no screen displays to worry about and certainly no coloured pixels. Extra BASIC keywords had to be invented to enable users to handle all these new pleasures, but software engineers on the design staff of each company had their own independent ideas on the syntax of these keywords.

Although the philosophy of free competition has much to commend it, it tends to contribute little to standardisation. On the contrary, it would

appear that manufacturers take some pride in the 'superiority' of their own version of BASIC over rival versions. It is impossible to be superior to something unless it is also different so BASIC has evolved in an undisciplined manner. In fact, there is no guarantee that a program will even remain compatible between *identical* machines if one is a much later version off the production line. It is not uncommon for a manufacturer to bring out a modified version only a few months after the launch of a new machine. These changes can take the form of improvements such as the addition of extra BASIC keywords or perhaps some slight alteration in syntax of existing keywords. However, established computer manufacturers are always conscious of the responsibility they owe to writers of software and normally ensure 'upward compatibility' between the old and new versions. That is to say, they arrange that programs written for the old version will continue to work on the new but not, necessarily, the other way round.

The speed of BASIC

Another disadvantage of BASIC is that it is slow in execution. Even its strongest devotees would admit this although, as they rightly point out, many programs appear to execute 'instantaneously' so what does it matter anyway? However, in certain areas, the slowness of BASIC can be irritating almost to the point of unacceptability.

If you are new to computing (perhaps the Commodore 64 may be your first computer) the speed at which the machine can calculate reams of square roots, trig functions and advanced mathematical relations can blur the critical faculties. Waiting about twenty or thirty seconds for a program to sort out a hundred or so names and addresses in alphabetical order is initially borne with equanimity. After all, it is still very much faster than a fellow human could do it. But, after a few months, tolerance and wonderment gradually give way to irritation and discontent. A stage is reached at which a computer is compared with other computers rather than with a human! Waiting for a blank screen to deliver the goods is about as exciting as watching grass grow. The trouble with BASIC is due to the fact that it is interpreted rather than compiled. As many readers will already be aware, the intrinsic language of any computer is machine code so the ability to program in any other language must depend on translation software. This must be available in addition to the resident operating system. To gain a clear idea of the difference between interpreters and compilers it is necessary to distinguish 'source code' from 'object code'. *Source code* is the program written in a particular 'high level' language (such as BASIC, COBOL, or FORTRAN) and *object code* is the machine code translation. Source code may be considered as the *input* to the translation software and object code as the *output*. The difference between compiling and interpreting can now be explained.

Compilers

These first translate the entire source code into object code, the process being known as 'compiling'. Only after compiling can the program be executed (run). Compiling takes time, in fact a relatively long time, but the program has to be compiled only once.

Once compiled, the original source code could, in theory, be thrown away (although this often demands more courage than most of us have). The program is now stored as object code which, as explained above, is machine code. This means that henceforth it will run like lightning whenever it is required. The essential idea behind a compiler is really that of a time shift. We put up with a lengthy translation time (which only has to be tolerated once) in return for fast execution of all subsequent runs.

Interpreters

These are a different kettle of fish altogether. Each line of source code is translated and then executed *before* the next line is translated so there is no tidy distinction between the source and object codes. When we instruct such a program to 'run', we are unconscious of the fact that each line has to be translated to machine code before the line is executed and that this takes place again on each subsequent occasion the program runs.

This wasteful re-translation each time is the fundamental reason why interpreted languages, such as BASIC, are much slower in execution than compiled languages. However efficient the interpretive software is written, there is always this same time penalty. However, in spite of this, there are advantages. During the original development of BASIC, the overriding consideration was to produce a language that would not only be easy for engineers, physicists and other non-specialists to use but, more importantly, easy to correct mistakes and/or change bits of the program at short notice. It is certainly easier and quicker to correct mistakes in an interpreted language because it is still in source code form.

Structured programming

Naturally the mistakes have first to be found. How easy it is to find them depends to some extent on the skill and experience of the programmer and also to the degree in which the language is 'structured'. It would be out of place in this book to discuss the ins and outs of 'structure' in great detail. Many books, and hundreds of magazine articles have already done justice to the art of structured programming. It is sufficient, here, to describe it as a programming style in which great stress is placed on the concept of *self-sufficient blocks of code*, each capable of independent verification. The

blocks, or 'modules' are strung together in a manner which simplifies final testing and, what is equally important, allows *future* modifications or additions to be incorporated without altering *existing* modules.

Some languages, PASCAL and ALGOL in particular, are designed in a way which forces programmers to write good structure. BASIC, at least the version used in most microcomputers, including the Commodore 64, has no such pretentions. In fact, BASIC has been subjected to much abuse for its complete lack of structure. It is said to encourage sloppy, ragged programming. Serious attempts have been made over the last few years to improve BASIC and some of the new versions have already incorporated some of the syntactical forms which, previously, were exclusive to the true structured languages.

Why learn machine code?

There are many reasons why you should make an effort to learn machine code and we shall, of course, be discussing some of them. Initially, the important question is 'When is the best time to learn it?' For example, do we buy a computer and then start right away learning machine code or do we first pass through some form of 'apprenticeship' in BASIC and then go on, as it were, to better things.

The answer to this depends to some extent on your strength of character. If you are easily frustrated and eager to get results then you would be well advised to stick to BASIC, at least for a time, in order to gain familiarity with the general principles of programming. Unfortunately, the longer you remain hooked on BASIC, the more difficult it becomes to leave it. If, on the other hand, you intend to become seriously involved in various aspects of computing, including the writing of software for commercial purposes, it would be wise to combine machine code studies together with BASIC from the start, a reasonable study ratio being about five to one in favour of machine code. Here are some of the reasons for learning it:

Speed
Programs written in machine code can be between one hundred to one thousand times faster than the same program written in BASIC.

Compatibility
Subject to a few provisos, programs written in machine code will run on any computer which uses the same microprocessor. In other words, machine code programs are specific to the particular microprocessor used in the computer rather than the computer itself. The provisos are that due regard must be taken of changes in the overall *memory map* of the computer and possible variations in *syntax* of the assembler used for entering the code. However, it is worth pointing out that an assembler is only an aid to the writing of code. It is possible to write machine code

programs on any computer by 'poking' code directly into memory cells. This method is tedious but the end result is exactly the same as when an assembler is used.

Machine knowledge
A machine code environment stimulates interest in both the hardware and system software. The microprocessor takes on a new look. It ceases to be what the media are pleased to call a 'miracle chip'. It can no longer be taken for granted because programming in machine code demands fairly intimate knowledge of the registers within the chip together with the interconnecting buses.

Peripheral projects
Few users of microcomputers devote much time to the input/output end. Such lack of interest places users at the mercy of the 'add-on' manufacturers. They remain content to buy ready-made devices that simply plug into the back. No disrespect is intended towards these manufacturers. In fact, they play an almost indispensable role in the provision of interfacing black boxes. Nevertheless, it is much more interesting if you have some idea of the theory behind interfacing, and can pay dividends in cases where the commercial project requires some slight modification in order to satisfy local conditions. Machine code work tends to force an interest in the input/output chips (such as the CIA in the Commodore 64) as well as the microprocessor.

Ego-boosting
This is trivial but we are all human. 'Experts' in machine code tend to be treated with just a tinge of respect, probably due to the mistaken belief that machine code can only be understood by those with exceptional intellects. It is rewarding to keep this myth alive although, inwardly, you soon learn that it is more to do with patience and tenacity than intellect. The average man or woman has more than sufficient intellect to master machine code but not all are able to muster or sustain sufficient interest!

How difficult is machine code?

Machine code is more difficult to use than BASIC but not greatly so. In fact the difficulty factor is much less than is popularly supposed. It is tedious rather than difficult. Unlike high level languages such as BASIC, each instruction is of atomic rather than molecular dimensions. Some instructions do little more than move eight bits from one place in the computer to another. Some may alter the arrangement of the bits within a location or register. Some are able to perform the primitive arithmetic processes of addition or subtraction. There are, of course, instructions which alter the sequential address flow of the form 'branch if' but they have no built-in sophistication such as the IF/THEN statements of BASIC.

Not very much happens when we write a machine code instruction. The first habit to acquire, and one which many people find the most difficult, is

the ability to think low enough! The relatively advanced style of communication we use to communicate with each other tends to spoil us. It is essential to get the brain into the lowest gear possible because although you need high power it has to be released at a slow rate! We have to face the fact that the microprocessor, the so-called 'miracle chip', has the mental capacity of an earthworm. Once this is realised, and firmly established, many of the difficulties vanish.

It may be argued that such a moronic view of a microcomputer is valid even when programming in BASIC. This is true in principle but not in magnitude. We should remember that the team that wrote the BASIC interpreter for the Commodore 64 injected some measure of human intelligence into the ROM. Syntax and other errors committed by programmers are gently pointed out by the operating system. In pure machine code, which can only be entered into the Commodore 64 by poking directly into RAM locations, there is not even a suspicion of humanity. Enter a wrong code by mistake and the system crashes. The microprocessor has no built-in intelligence and offers no advice whatsoever. Only codes it can understand are obeyed. Leave one tiny loophole open in, say, an end-of-loop test and the microprocessor falls straight into it. If code is entered by an *assembler*, such as MIKRO, there is some improvement in friendliness, although this is due mainly to less error-prone methods of entering code.

Commodore 64 hardware

It is wise for prospective machine code programmers to develop an interest in the hardware side. This does not mean spending long hours studying electronics. It is strange, and rather disconcerting for mature engineers and technicians, that modern computers, although almost entirely electronic in nature, can be understood quite well by those with only a smattering of electronic theory. Silicon chips have displaced the conventional theoretical circuit. The complete schematic diagram of the Commodore 64 contains only eight discrete (separately manufactured) transistors. In contrast, the number of transistors integrated within the chips would probably reach a figure in excess of 100000.

Understanding and repairing microcomputers is no longer the exclusive province of highly trained electronic engineers. An ability to think logically is the primary quality and more important than theoretical electronic knowledge. The advantages of academic qualifications in electronics is, rather sadly, slowly being reduced to the status of an optional extra.

The MPU

The Micro Processing Unit controlling the Commodore 64 is a 6510A microprocessor, occupying position U7 on the circuit board. The 6510A is

virtually identical to the more widely known 6502 chip, manufactured by MOS Technology. Both these chips are, in fact, software-compatible with each other and both share the same instruction set. The full details of the microprocessor will be found in Chapter 3 but, because this book may also be read by other than Commodore 64 users, it was felt that it should continue to be referred to as a '6502'.

The 6510A has an extra facility in the shape of an on-chip input/output port. Although eight input/output lines (P0 to P7) are provided, only six of these are used. Three of them are dedicated to the cassette recorder and three for switching in different memory chips. The machine code programmer will normally have little interest in this port because it remains obscurely in the background and is controlled entirely by the operating system. Further details can be found in the Commodore 64 Programmer's Reference Guide.

On-board ROMs

The permanent (resident) software in the Commodore 64 is buried, quite logically, within five separate ROM chips. The individual chips are now described.

The operating system ROM

The operating system ROM is called the 'kernal' by Commodore and has a capacity of 8K bytes and the type number 2364A. It occupies position U4 on the circuit board. The range of hexadecimal addresses allocated to the kernal is E000 to FFFF (59344 to 65535 decimal). It lies, therefore, at the extreme top of the 64K memory map. The software buried within the kernal ensures that the conflicting demands of various sub-units in the computer are dealt with in a calm and logical fashion and with priorities normally dependent on their relative speeds. Controlling the keyboard and display system is an important responsibility which is left to the kernal.

The BASIC ROM

This chip contains the interpreter software necessary for users who wish to communicate with the computer via BASIC. Like the kernal, the chip type number is 2364A and it occupies position U3 on the circuit board. The ROM has a capacity of 8K bytes and the address range allocated to it is A000 to BFFF hex (40960 to 49151 decimal).

It is difficult to write a BASIC interpreter that will fit into the tight confines of an 8K ROM. The software writers have done their best but there is no point in denying that the Commodore 64 will never be renowned for the quality of its BASIC. It must be considered a primitive version of the language without frills or fuss. Because of this, there is a greater need for, and a greater urge to learn, machine code in order to write subroutines to supplement BASIC.

The character generator ROM

When a character is displayed on the screen, it is not immediately obvious that we are looking at a *pattern of dots*. In fact, the principle behind the display of a screen character is similar to the principle used in a normal dot matrix printer. Each character has its own unique pattern of dots. A single dot on the screen can be produced by a single bit in the '1' state and the absence of a dot by the same bit in the '0' state.

Each character in the Commodore 64 is formed on a background of 64 dot positions, arranged in an 8×8 matrix. In other words, each character requires 8 bytes of information. The patterns for 512 different characters are stored in the character generator ROM in the form of two character sets. One set of 256 characters covers upper-case characters and fixed keyboard graphics and the other caters for the more normal 'typewriter style' upper- and lower-case characters. The ROM carries the type number 2332A and occupies position U5 on the circuit board.

The character ROM only generates characters. To display a character, the operating system arranges that the chosen character in the ROM is placed in that part of RAM designated as 'screen memory' which (normally) occupies the decimal address range 1024 to 2023. Any character pattern from the ROM placed in this area will be displayed at a position determined by the current cursor or by means of a direct poke.

The RAM chips

The Commodore 64 uses, as its name suggests, 64K bytes of user RAM. This is provided by a bank of eight chips, occupying positions U12,U24,U11,U23,U10,U22,U9 and U21. All eight chips are identical, bearing the type number 4164–2. Each chip has a capacity of 64K bits and therefore can only contribute a *single bit* towards a memory byte. From the user's viewpoint, the RAM appears as one chip with a capacity of 64K bytes because the address selection wires of each chip are common to all eight.

It is interesting to note that the chips only have 8 address lines labelled A0 to A7 and yet, to provide 64K (2^{16}) different addresses, the law of binary combinations demands that at least 16 address lines are required. However, pins on a chip cost money and the more there are, the more intricate (and costly) is the final circuit board which holds them. It has been common practice for some years to compromise on the number of address lines by supplying the address in two equal instalments. The addressing matrix within each 64K chip is arranged in eight columns and eight rows. Only eight address lines, marked MA0 to MA7, are required to feed the chip because control lines (CAS and RAS on the chip) switch the first batch of 8 to the row address and the next batch to the column address. The steering of each 8-address bit from the 16-bit microprocessor address bus is achieved by two 74LS257 multiplexer chips. A 'multiplexer', in this sense,

is an electronically operated multi-arm 'switch' without any moving parts.

Memory organisation

One of the awkward features of the 6510A, in common with almost all other popular 8-bit microprocessors, is the upper limit imposed on directly addressable memory due to having only a 16-bit address bus. The number of possible binary combinations (the number of unique addresses) is therefore 2^{16} which is 65536 and, because 1K=1024, is 64K.

When 8-bit microprocessors were first launched in 1971, semiconductor memories were very expensive so it was probably considered at the time that a 16-bit address bus, supporting 64K of memory, would be more than ample. It was never envisaged that microcomputer users would ever be able to afford, or even need, anything like 64K of memory. They were wrong. In the last few years, the price of semiconductor RAM has fallen sharply, due to volume production and advances in production technology.

We now come to the strange case of the Commodore 64. It supports the full 64K of RAM which, according to our previous remarks, completely fills the directly addressable memory space. But there is, in addition, 20K of ROM which means that a total of 84K of addressable space is needed. Commodore solved this problem by choosing the 6510A microprocessor instead of the 6502. You will remember that the 6510A has a built-in 8-line input/output port. Three of these lines P0, P1 and P2 are used to switch the three ROMs in and out of the available address space. Although this switching action normally rests in the *default* mode by the operating system, there may be occasions when the machine code programmer will want to alter the sequence in order to cater for a special situation. For example, the normal (default) situation for memory distribution is as follows:

(1) 38K of RAM available for user's programs.
(2) The 8K kernal ROM and 8K BASIC ROM.
(3) The remaining 10K is RAM used by the operating system for input/output buffering and as general-purpose 'workspace'. If the 32K of user RAM is not sufficient, it is possible to alter the memory distribution to obtain more RAM but this naturally means sacrificing some of the facilities. However, for the benefit of those who may wish to alter the default system, the manner in which the switching is carried out will now be described.

ROM/RAM switching

In common with most input/output ports, the data lines can be programmed to behave as inputs or outputs, depending on the bit pattern

placed in the Direction Register. The data lines themselves are controlled by the Data Register. These two registers have the following fixed addresses:

Direction Register	0000 hex
Data Register	0001 hex

To make any particular data line behave as an output, the corresponding bit in the Direction Register must be programmed with a '1'. To ensure input behaviour, the corresponding bit must be a '0'. Figure 1.1 shows relevant details of the two registers and the ROM/RAM switching lines.

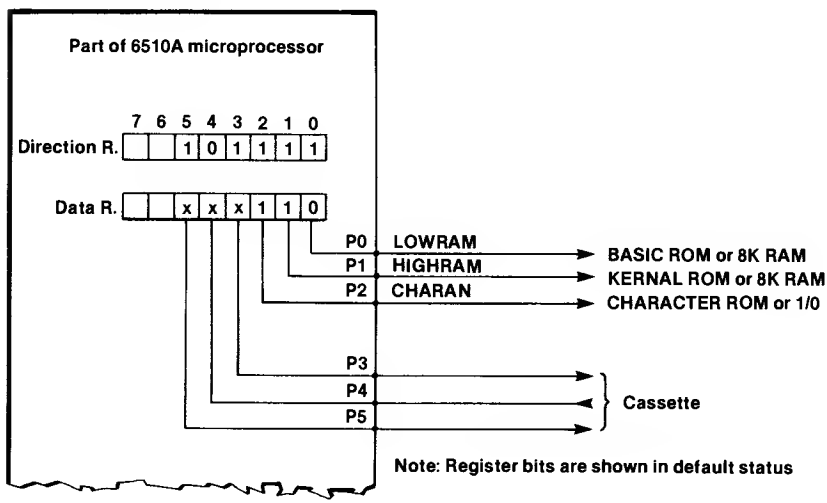


Fig. 1.1. Memory switching by input/output port lines.

The *Direction Register* bits are initialised to 0010 1111 by the operating system on power-up. Since the three *rightmost bits* are '1's, the lines P0,P1 and P2, which control the switching, behave as outputs. The remaining bits are of no concern at this stage because they are used as control lines for the motor and sense circuits of the tape cassette.

The *Data Register*, which controls the logic state on P0,P1 and P2, is normally programmed by the operating system. The sequence is as follows:

(1) P0 drives the signal called LORAM which switches the BASIC ROM and normally rests in the HIGH state (the '1' state). When this line goes LOW (the '0' state), the ROM is switched out of the memory space, leaving 8K of addressable RAM in its place over the range A000 to BFFF hex.
(2)P1 drives the signal called HIRAM which switches the kernal ROM and normally rests in the HIGH state. When this line goes LOW, the ROM is switched out of the memory space, leaving 8K of addressable RAM in its place over the address range E000 to FFFF hex.
(3) P2 drives the signal called CHARAN which switches the character

Generator ROM and normally rests in the HIGH state. The ROM occupies the same address space as the input/output (I/O) devices (D000 to DFFF hex) but only replaces them when CHARAN is driven LOW.

It is clear from the above that the default state of P0,P1 and P2 are all initialised to the '1' state by setting the three rightmost bits in the Data Register accordingly. The memory distribution can be altered to suit individual requirements by adjusting these bits. Care should be taken to preserve the state of the remaining bits because they are used for the cassette unit.

Cassette unit control

The three data lines P3,P4 and P5, on the 6510A input/output port, control the Commodore C2N cassette unit. Figure 1.2 shows the connections. The following functions are performed:

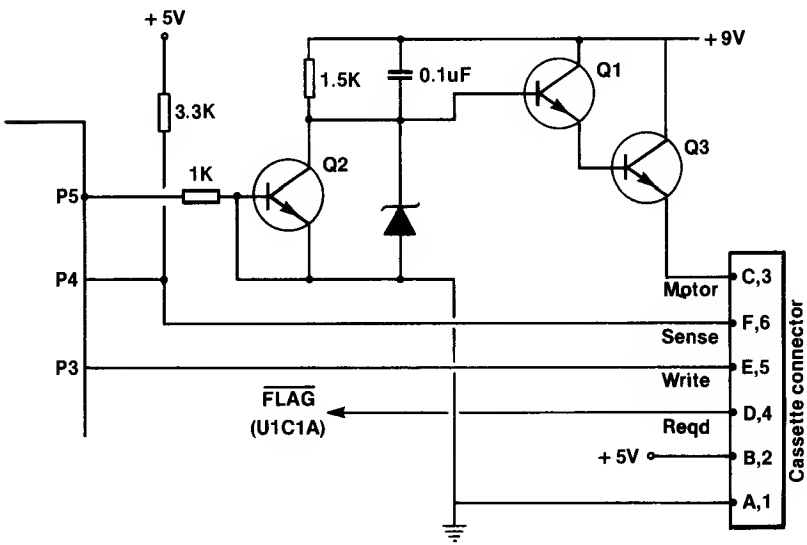


Fig. 1.2. The cassette control circuit.

(1) P3 is an output to the CASS WRT line. It goes direct to pin E5 on the 6-pin male edge connector and carries data to be written to tape.
(2) P4 is an input and receives the CASS SENSE signal from pin F6 on the edge connector. It senses if the PLAY button on the cassette unit has been pressed and, if not, sends the appropriate error message to the user.
(3) P5 is the CASS MOTOR output but is quite incapable of delivering sufficient current level for directly driving the cassette motor. Amplification is provided by three transistors Q1,Q2 and Q3. The first stage, Q2, is a

conventional *ground-emitter* amplifier delivering an inverted output signal to Q1 and Q3. These are connected as a *darlington pair* (see Chapter 8) to ensure high current gain. When P5 goes LOW, Q2 is cut off, causing the collector voltage to rise. This, in turn, drives the darlington pair into full conduction and completes the 9-volt circuit to the cassette motor. The information read from the cassette tape arrives at pin D4 on the 6-pin edge connector. It then connects to pin 24, the FLAG input, of the CIA located at position U1 on the circuit board.

The CIA chips

There are two CIA (Complex Interface Adaptor) chips on the circuit board. They are specially designed to cater for a wide range of input/output requirements. They bear the type number 6526 and are not unlike the more familiar 6522 VIA chip (Versatile Interface Adaptor). The CIA provides two separate 8-bit input/output ports (known as side A and side B) with the customary pair of handshaking controls on each port. There are 16 addressable registers including direction and data registers, timers, counters and control registers.

The details of the CIAs will be left until Chapter 8 since they are sufficiently complex to deserve separate treatement. However, it is worth noting here that the CIA occupying position U1 on the circuit board is, more or less, dedicated for reading the rows and columns of the keyboard. It sits in the 16 locations DC00 to DCFF hex, on the memory map.

The B side of the CIA occupying position U2 on the circuit board provides the facilities of a user port. It sits at the address range DD00 to DDFF hex in the memory map. The port is available from the 24-pin edge connector CN2.

The Video Interface Chip

This is known as the VIC-II chip. It contains the truly complex circuitry for producing the low and high resolution displays besides allowing easy control of the little moving objects which have been given the rather delightful name of 'sprites'. It bears the type number 6567 and occupies position U18 on the circuit board. It has 47 internal registers addressable within the address range D000 to D02E hex. It was designed as a general-purpose colour video control chip for video terminals or video games.

The Sound Interface Device (SID)

This is virtually a complete music synthesiser on a single chip. SID occupies position U18 on the circuit board and bears the type number 6581. It contains 29 programmable registers, addressable over the range D400 to D41C. Full details of the register functions in VIC and SID are contained in the Commodore Programmer's Reference Guide.

Summary

1. Source code is the program as entered in high level or assembly language and is the input to the translation software.
2. Object code is the output from the translation software and is in a form which can be directly executed.
3. A compiler is software which translates the entire source code into object code, ready for execution.
4. An interpreter is a program which translates each line of source code into object code then executes it before translating the next line.
5. Once a program is compiled, the original source code is no longer required.
6. Programs which are interpreted (such as BASIC) always require the presence of source code.
7. Compiling takes time but only has to be done once, so all subsequent execution takes place rapidly.
8. Machine code programs are microprocessor-, rather than machine-, compatible.
9. The Commodore 64 uses the 6510A microprocessor which has an on-chip I/O port. Otherwise, it is software-compatible with the 6502.
10. The 8K operating system ROM is called the 'kernal'.
11. 64K of RAM is provided by eight 64K one-bit dynamic RAM chips, although normally, only 38K of this is on-line.
12. The BASIC and other ROMs can be switched out of the memory map if extra RAM is required.
13. ROM/RAM bank switching is dependent on the states of P0,P1 and P2 terminals on the microprocessor.
14. The data register, for setting the states of P0,P1 and P2, is addressed at $0001.
15. The cassette interface is driven from P3,P4 and P5 on the 6510A I/O port.
16. There are two Complex Interface Adapator chips (CIAs). One of them is dedicated to the keyboard and control ports and the other is mainly for the user port and the serial interface.
17. Video effects are handled by the Video Interface Chip (VIC).
18. Sound synthesiser effects are handled by the Sound Interface Device (SID).

Self test

1.1 Is a program, written in BASIC, source code or object code?

1.2 State two advantages of compiling over interpreting.

1.3 State two disadvantages of compiling over interpreting.

1.4 Name two languages which were designed with the needs for structured programming in mind.

1.5 State the main difference between a 6502 and a 6510A microprocessor.
1.6 What is the name given by Commodore to the operating system?
1.7 If a ROM occupies the address range &D000 to FFFF, how many kilobytes are stored?
1.8 How many dots are used to form a screen character and how is the matrix arranged?
1.9 State the normal top and bottom addresses of screen memory.
1.10 How many bits are stored in one of the RAM chips? (Answer in decimal.)
1.11 Why is a multiplexer necessary in the RAM address system?
1.12 How much RAM is available to the user without sacrificing some of the normal facilities?
1.13 The normal (default) distribution of RAM and ROM can be changed by altering the bits in a certain address. What is this address?
1.14 How do you gain an extra 8K of RAM at the expense of the kernal?
1.15 Which particular I/O port is responsible for the cassette interface?

Chapter Two
The 6502/6510A Microprocessor

Abbreviations and conventions

There are many of these – in fact, too many for comfort. Although a comprehensive list of terms is explained in the glossary (found at the end of this book), the following selection will be found particularly relevant when reading this and the following chapter.

---

lsb=least significant bit. msb=most significant bit.
Bit positions within a byte are numbered 7 6 5 4 3 2 1 0.
Bit 0 is the lsb. Bit 7 is the msb.

A=the accumulator. X=register X. Y=register Y.
P=process status register. PC=program counter. PCL=low byte of PC.
PCH=high byte of PC. SP=stack pointer. ALU=arithmetic and logic unit.
AR=address register. ARL=low byte of AR. ARH=high byte of AR.

Process status flags:
N=negative (bit 7). V=overflow (bit 6). B=break (bit 4).
D=BCD (bit 3). I=interrupt (bit 2).
Z=zero (bit 1). C=carry (bit 0).

Although the microprocessor used in the Commodore 64 is a 6510A, it is completely compatible with the more popular 6502. To reassure readers who already have some experience with the popular version, we shall still refer to the microprocessor as a 6502. It is possible to plunge straight into machine code programming without troubling too much about the technical details of the 6502 or 6510A. However, it pays dividends in the long run if some of the internal behaviour is understood and it can also be interesting for its own sake.

The 6502 family versus the Z80

The 6502 and the Z80 8-bit microprocessors have retained their popularity with personal computer manufacturers for many years. Their popularity is likely to remain until the approaching 16-bit revolution is established. Both the 6502 and the Z80 have good and bad features which are fairly equally distributed. The Z80 has sometimes been praised as the more powerful of the two but, in the absence of a satisfactory definition of 'power' that praise has little substance. If by 'power' we mean execution speed then neither is superior to the other. Some types of program can execute faster on the Z80, others execute faster on the 6502. Because of this it is not wise to pay too much attention to 'benchmark' tests.

The Z80 has a powerful marketing advantage because of its downward compatibility with the Intel 8080. The widely used disk operating system CP/M, for which an enormous amount of commercial software has been written, is based on the 8080 instruction set so any microcomputer which runs on the 6502 could be said to be disadvantaged in this respect. However, this should not trouble us too much because CP/M has not been without its critics. What popularity it used to enjoy was due to the fact that it was the only disk operating system around at the beginning of the microprocessor era. An enormous amount of software has been written to run under CP/M which accounts for its still being around.

6502 architecture

As most readers will already be aware, programs written in machine code for any given microprocessor should, subject to minor variations, still run on any make of computer employing the same microprocessor. That is to say, machine code programs are microprocessor- (rather than machine-) specific. The 'minor variations' mentioned above include such things as differences in the way memory is allocated – in particular, the amount and location of free space. Machine code programs are usually written with the aid of an *assembler* and some variation in syntax can be expected between different commercial versions.

It is better to begin by reviewing the microprocessor in relation to other main components of the system as shown in Fig. 2.1. The microprocessor communicates with the rest of the computer via three bundles of wires known as *buses*. The *address bus* is responsible for picking out the particular memory location required by the programmer. The *data bus* is responsible for sending or receiving data to and from the chosen location. The *control bus* is a hotch-potch of wires, necessary for the overall discipline of the system.

Note that the address bus is shown split down the middle because it is important always to bear in mind that a 4-hex digit address code is handled

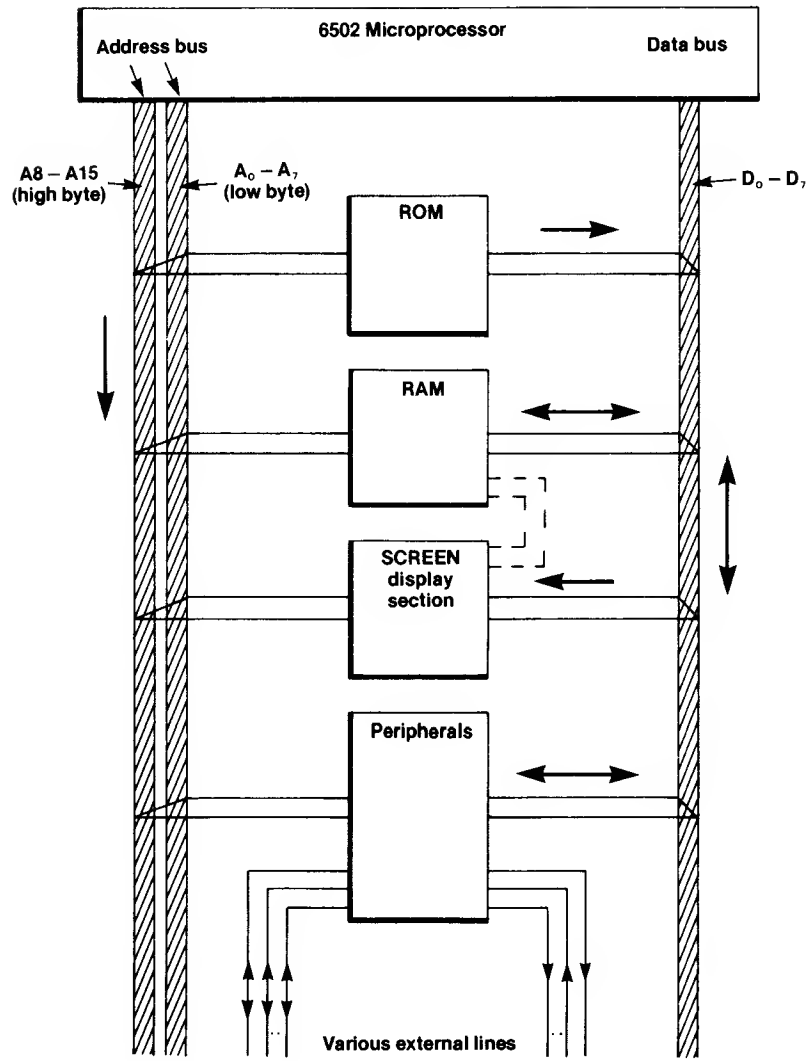


Fig. 2.1. Position of the 6502 in relation to the external bus devices.

by the microprocessor in two halves, the lines A0 to A7 (high byte) and lines A8 to A15 (low-byte).

The ROM chips

These contain fixed information and cannot be subsequently altered by the computer. Notice from Fig. 2.1 that the data flow arrow is one-way only, indicating that data can only flow from the ROM to the 6502 via the data bus. The information stored includes the 8K operating system of the computer (CBM call this the 'kernal' ROM). The BASIC language

interpreter is also an 8K ROM. The most important characteristic of ROMs is the permanence of the stored information which is retained after the power is disconnected.

The RAM chips (Random Access Memory)

The title is misleading because the essential quality of RAMs, which distinguishes them from ROMs, is the ability to *change* the stored information under program control. The mere fact that they are 'random' access is incidental because so also are ROMs. In other words, RAMs are really *read/write* memories.

Depending on the internal structure, RAMs may be further classified into *static* or *dynamic*. Some writers refer to dynamic RAMs as DRAMs, the 'D' prefix standing for dynamic. Due to the need for reducing current consumption and maximising packing density, each bit is stored within the inter-electrode capacity of MOS transistors. The stored information, however, is a transient affair because it is only a minute electrostatic charge which leaks away in a few milliseconds. Consequently, each stored bit must be periodically recharged in order to compensate for the leakage. This process, called 'refreshing' is inherent in the hardware design and is not the responsibility of the programmer. However, the refresh-cycle takes up extra time. Dynamic RAMs are therefore a compromise in which access time is sacrificed in order to increase packing density and reduce cost.

The Commodore 64, and indeed nearly all other makes of microcomputers, will use dynamic RAMs. The alternative would be to use static RAMs but the cost would be prohibitive and they would occupy a greater space on circuit boards. From now on in this book, the term RAM will be taken to mean the dynamic type. Notice from Fig. 2.1 that, unlike ROMs, the RAMs are fed by a bidirectional data bus. The data flow arrow indicates that data can be passed either way between the memory locations and the microprocessor.

6502 systems are *memory-mapped* so it is not surprising that the keyboard, screen display and the input/output interfaces are strung across the address and data buses as if they were memory chips. In the case of the screen display, the dotted line on the figure indicates the additional data path between the area of RAM dedicated to the screen and the display circuits. To avoid cluttering the diagram the various signal lines, forming the 'control bus', are not shown.

Inside the 6502

Figure 2.2 shows reasonable, but by no means complete, details of the paths between the various registers. Such paths within a microprocessor were often called 'highways' because they ramified over the chip area, providing a kind of long-distance communication.

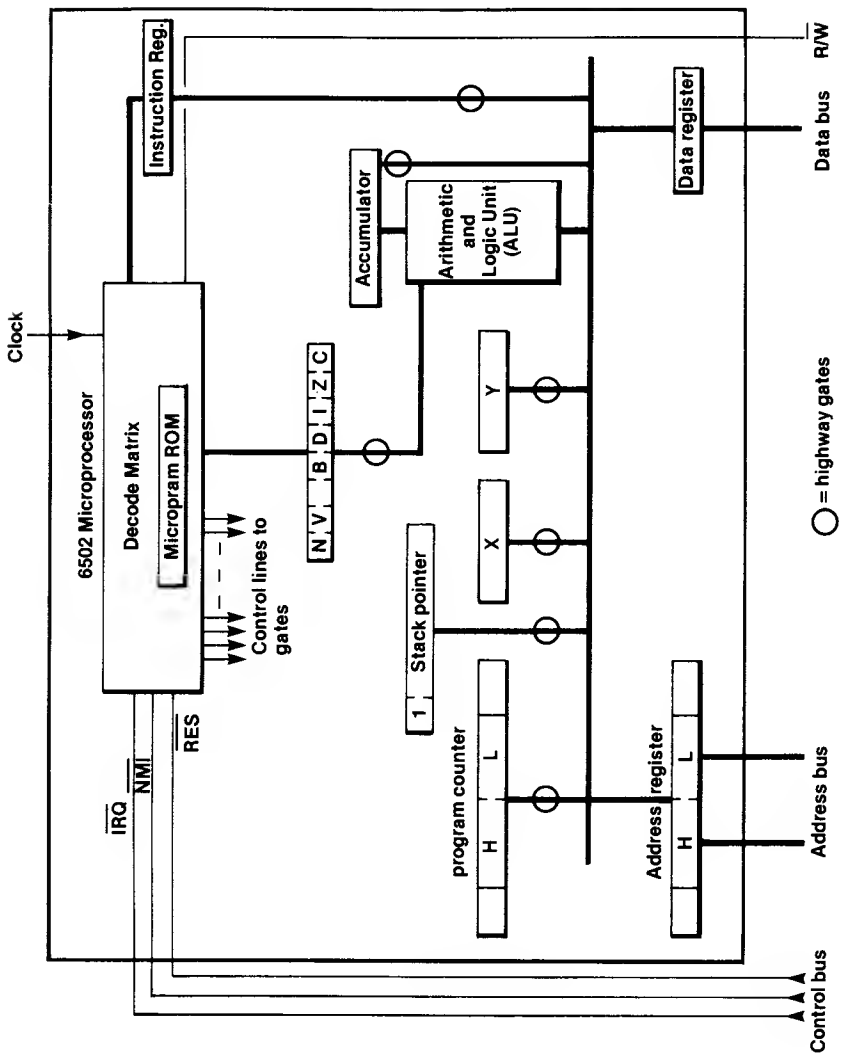


Fig. 2.2. 6502 registers and highways.

Most of the units within the microprocessor are 8-bit registers, the exception being the Program Counter which is 16 bits wide. Control lines (not shown) operate the input and output gates of each separate register, ensuring that only *one pair* is allowed access to the highway at any one time. For example, during the machine code instruction TAX (which means Transfer Accumulator to X register) only register A output gate and register X input gate are open to the data highway. This makes the highway free to pass the contents of A to X without being jammed by data resting in any of the other registers.

The majority of instructions we give to microprocessors are in the nature

of data *transfers* either between internal registers or between registers and external RAM,ROM or peripherals. Some instructions, such as ADC (ADd with Carry) perform arithmetical operations on the data but this still has to be fetched from somewhere. Even a simple instruction like INX (INcrement contents of X) involves a transfer because the X register is not equipped for altering itself. Instead, the contents of X must be transferred along the highway to the arithmetic section before the 1 can be added. It then has to be returned.

Directly programmable internal registers

It is assumed that many readers will already be aware of the various registers and their functions but, for the sake of continuity, a brief description follows together with the standard abbreviations subsequently used in all references. A distinction is made between directly programmable and other registers which, although playing a vital role, remain in the background, unseen by the programmer.

Accumulator (A)

This register has a supreme role. It is the only one capable of performing arithmetic processing. This is evident from Fig. 2.2 which shows that, in addition to the usual connection to the highway, there is a direct and exclusive link to the Arithmetic and Logic Unit (ALU). It is involved in transfers to and from memory and acts as an interim data storage during arithmetic and logic operations.

For example, during a simple addition of two numbers (ADC), the first number must pass to the accumulator and is then 'entered' to a holding register within the ALU. The second number then enters A, the addition is carried out and the result sent back to A. Those used to scientific calculators in the Hewlett Packard range will recognise the inherent Reverse Polish (RP) action. It is worth digressing a little to explain RP. A Polish mathematician proposed a new method of expressing arithmetic, the essence of which was placing the operator (+,−,×, etc.) after, instead of in between, variables. For example, instead of writing A+B to indicate addition, he proposed that it should be written as AB+. Because his name was quite impossible to pronounce, his system has become known simply as Reverse Polish Notation (RPN or simply RP). Von Neumann, who is often referred to as the father of the modern computer, suggested that the arithmetic system of digital computers would operate most efficiently if based on RPN. Thus the ALU in the 6502, in common with nearly all other microprocessors, require the two variables first, the add operator is then activated and the result passed to the accumulator, replacing the previous contents.

The dominance of the accumulator over other registers is evident from

the instruction set of the 6502. However, the fact that only one accumulator is present gives ammunition for the protagonists of the rival Z80 which boasts eight accumulator type registers. A single accumulator does tend to be restrictive in organising efficient machine code.

The X and Y registers

Like the accumulator, the X register and the Y register (subsequently referred to as X and Y) are both 8 bits wide. They have three primary uses in programming:

(1) They make up for the inconvenience of the solitary accumulator. Important data residing in A can be transferred temporarily by the use of TAX or TAY and later, when A is free, transferred back using TXA or TYA.
(2) They can serve as up-counters or down-counters for setting up machine code loops. This is due to the ease by which they can be incremented or decremented by the instructions INX,DEX,INY or DEY. It is curious that the designers failed to provide an equivalent instruction for incrementing or decrementing A. The only way is to use the relatively inefficient method of adding or subtracting 1, using ADC or SBC.
(3) They are fundamental to the technique known as *address modification* by indexing. When using an indexed addressing mode (denoted in assembly form by a comma followed by X or Y), the data in the X or Y register is automatically *added* to the data in the operand. The resultant is interpreted as the address of the required data.

This idea was pioneered by a team at Manchester University and, at the time, represented a huge step forward in computer science. They called the index register the 'B box', presumably to differentiate it from the accumulator A. Before this, altering the operand address in loops was cumbersome. It involved loading the operand from inside the program, incrementing it and then storing it back in the original position. In other words, it was necessary to alter the program in order to modify an address. Indexed addressing is so much cleaner to work with and certainly less error-prone. Most of the indexable instructions in the 6502 allow a choice of using either X or Y for indexing. Although indexed addressing is dealt with later in detail, there is no harm in a little anticipation for the benefit of those who are new to the idea. So let's study the following example. Assume that X contains 30 and that we wish to use indexed addressing, writing, say, LDA 100,X. The simple instruction LDA 130 would have had the equivalent effect in that both would have loaded the contents of address 130 into A. The advantage of the indexed over the simpler form will be apparent when organising loops involving action on consecutive addresses.

This discussion should help to explain why the address bus, as well as the data bus, has access to the ALU. This should be understandable now it is recognised that the index register contents have to be added to the operand.

After all, address modification by indexing produces a *computed address* and only the ALU can truly compute.

The Process Status register (P)

If we define a register as an internal memory location for holding or processing data, then the Process Status register (P) is not a register at all. It is in fact a collection of isolated single-bit storage cells (flip-flops). Each bit is called a *flag* because it conveys certain information in yes/no form either for the benefit of the machine or the programmer.

After *most* instructions, the relevant flags are updated, depending on the result. There is no connection, either in the hardware or software aspects, between different flags. In spite of this, it is convenient and conventional to refer to it as a 'register'. It is important to the programmer to understand the exact significance of each flag – that is to say, under what conditions they are set or reset. It is also important to know which are under sole control of the microprocessor and which are directly programmable. There are seven bits in P, defined as follows:

The N bit: If this is 1, the last result contained a 1 in bit 7 position. The N bit is often misleadingly called the 'sign bit' because two's complement arithmetic recognises bit 7 as the sign rather than magnitude. If the number is unsigned binary, the N flag merely indicates the state of bit 7. It is automatically set or reset and is not directly programmable. BMI (branch if minus) and BPL (branch if plus) are the relevant branch instructions conditional on the state of the N bit. Most instructions leave the N bit updated as part of the execution routine, the notable exceptions being STA,STX,STY, TXA, and all branch and jump instructions. LSR is unique in that the N bit is always reset to 0, irrespective of the result.

The V bit: If this bit is 1, it indicates that the last arithmetic instruction caused two's complement overflow due to the result being outside the capacity of a single byte. It can be tested by the conditional branch instructions, BVS or BVC. The V bit is not important to the programmer if he is using unsigned binary because bit 7 of the result represents magnitude rather than sign. In this case, therefore, it can be ignored. However, the V bit also plays a major role in the BIT test instruction, assuming the same state as bit 6 of the data being tested. It is possible directly to clear the V bit to 0 by the instruction CLV although there is no corresponding instruction directly to set it to 1. Only the instructions ADC,SBC,BIT, PLP,RTI and CLV affect the V bit.

The B bit: This is set to 1 when a BRK instruction is encountered. Its significance is limited almost entirely to interrupt sequences. It cannot be directly programmed.

The D bit: The 6502 can perform arithmetic on straightforward binary numbers or on BCD (Binary Coded Decimal) numbers. The programmer

decides this by the use of either SED (Set Decimal) which makes D=1 or CLD (Clear Decimal) which makes D=0. The arithmetic mode currently in use *remains* until the D bit is altered. The default mode is D=0. The instructions which effect the D bit are CLD,SED,PLP and RTI.

The I bit: This is called the interrupt mask bit or the interrupt inhibit. It is inspected by the microprocessor when an interrupt request is received from a peripheral source. If it is 1, the request is not granted. It can be directly set to 1 by SEI (Set Interrupt) or cleared to 0 by CLI (Clear Interrupt). These instructions are vital when designing the software for peripheral interfaces, most of which will be interrupt driven. The instructions which affect the I bit are BRK,CLI,SEI,PLP and RTI.

The Z bit: This is the zero bit, and is set to 1 when a result is 0. This is worth emphasising strongly because it is often interpreted back to front. If a result is non-zero, the Z bit goes to 0. It can be tested by the branch instructions, BEQ (branch if equal to zero) or BNE (branch if not equal to zero). There are no instructions which can *directly* affect it. Most instructions affect the Z bit. The exceptions include TXS,STA,STX,STY and the branch and jump instructions.

The C bit: This is the carry bit, and is set to 1 when a carry out from the msb is detected. Instead of the bit 'dropping on to the floor' it is popped into the C bit. It can also be thought of as the ninth bit, particularly in shift and rotate instructions. It can be tested by the branch instructions BCS (branch if carry set) or BCC (branch if carry clear). It can also be directly programmed by SEC which sets C to 1 or CLC which clears C to 0. Instructions which affect the C bit are ADC,SBC,ASL,LSR,ROL,ROR, SEC,CLC,PLP,RTI,CMP, CPX and CPY.

It is clear from the above that the process status register flags have a profound effect on program behaviour. The majority of errors encountered, particularly when setting the terminating conditions for loop exit, are due to misinterpreting the behaviour of the flag bits. Unless you are already confident in this area you would do well to re-read the above treatment several times.

The stack pointer (SP)

This is an 8-bit register, dedicated to the automatic control of a special area in page one in RAM memory designated the 'stack'. Its function is in the nature of an *address generator*. It is impossible to describe the stack pointer fully without describing the stack itself. Because the stack is so important in its own right, discussion of its anatomy will be postponed. It is sufficient at this point to grasp the following essentials:

(1) The contents of SP is interpreted by the microprocessor as the address of the currently vacant location in the stack.
(2) To ensure that the address is always on page 1, rather than page 0, a

permanent 1 is hardwired at the msb end of SP acting as a ninth bit. Thus if SP itself contains 0000 0011 (0003 hex), the address is interpreted as if it were 1 0000 0011 (0103 hex) – that is to say, address 3 on page 1 rather than address 3 on page 0.

(3) SP can be initially loaded to any address on page 1 but the method is a little cumbersome. There is no actual instruction to load SP directly. It is necessary to first load X and then transfer it to SP by the instruction TXS. This may seem a chore but in practice it may only have to be done once, during the initialisation phase of a complete machine code program. In fact, it would generally be unwise to tamper with SP at all when using the assembler because it will have been initialised by the ROM operating system. However, if you are brave enough to attempt circumvention, SP is normally initialised to FF hex in order to utilise the entire stack area.

(4) Once initialised, use of the stack is simplicity itself. If you want temporarily to save the contents of the accumulator, without having to specify a storage address, just push it on to the stack with PHA (Push A). To retrieve it again, pull it back with PLA (Pull A). It is not possible to push X or Y directly but it can be done piecemeal by first using TXA or TYA.

(5) The stack is a LIFO, meaning Last In First Out memory so you must pull data back with this in mind. After every push, SP decrements by 1 in order for the next push to operate on a new vacant location. When data is to be pulled back, SP first increments by 1 (in order to point to the last stored item) before the pull operates. The stack pointer automatically 'rises' with each push and falls with each pull so there is no need to bother with SP (see Fig. 2.3). You can forget the existence of the stack pointer providing you remember that the last item pushed onto the stack from A will always be the first item pulled back into A.

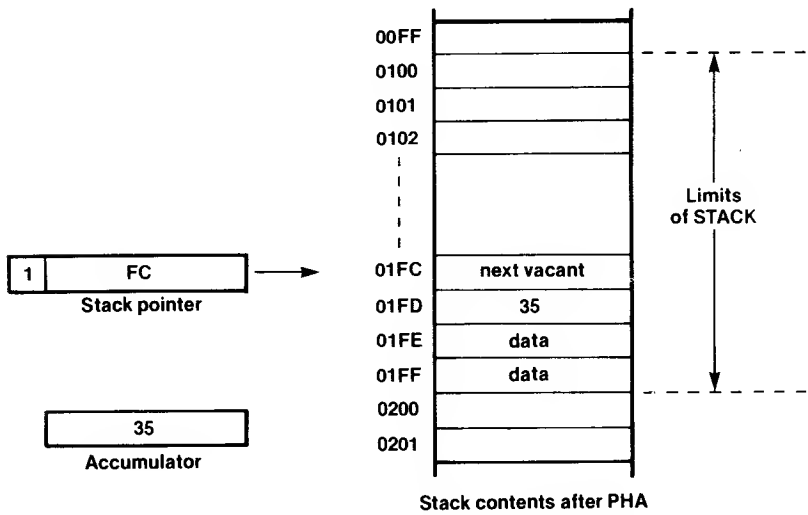


Fig. 2.3. The stack and stack pointer.

(6) The stack can only hold one page, i.e. 256 bytes. If you overflow the stack, there is no friendly warning as in BASIC. All that happens is a 'wrap around' effect. For example, if SP is initially set to FF hex and data keeps piling on the stack, SP eventually reduces decrements to 00. The next decrement wraps around to FF again causing weird and unexpected results. However, a 256 byte stack is normally more than ample for most programs and overflow conditions should be rare.
(7) In addition to its use as a temporary dumping ground for general work, the stack plays a vital role in both subroutines and interrupts. When a subroutine is called by means of JSR, the two bytes forming the return address (which will be in the program counter) are pushed onto the stack – high byte first, low byte second. When the subroutine ends (with RTS), the return address is pulled back from the stack (low-byte first, high-byte second) and passed to the program counter, allowing the body of the program to resume again.

Registers which are not directly programmable

In any microprocessor, some of the most important registers remain transparent (or at least translucent) to the programmer. That is to say, instructions are not provided which make direct reference to them. In fact, the more important a register, the less likely it is that the programmer is allowed direct access. In the 6502, the unseen registers (refer back to Fig. 2.2) are the Program Counter (PC), the two address registers ADL,ADH and the Instruction Register (IR).

The Program Counter

This enjoys the honour of being the only 16-bit register in the 6502. If there is an established register hierarchy, then PC is the undisputed candidate so its function deserves strong emphasis:

The Program Counter is a 16-bit register which always contains the address of the *next* instruction byte to be executed. The 16-bit length allows reference to any address in the entire 64K range.

Once a stored program is commanded to 'start execution', the following automatic sequence begins:

(1) The contents of PC is transferred to the address bus and the first instruction byte at that address is loaded into the computer and 'processed'.
(2) The PC then increments by 1.
(3) The PC is again transferred to the address bus and the next instruction byte is loaded and processed.

The sequence continues indefinitely, sweeping through the program bytes like a scythe until it is halted legitimately or reaches an illegal code. The sequence makes *no distinction whatsoever between program and data.* It is up to the programmer to arrange the instruction bytes in consecutive address order and organise either a break (BRK) or an orderly return to the operating system loop. If the PC is allowed to reach data bytes it will interpret these as instructions which the 6502 will either attempt to execute or crash.

It is all very well describing the sequence but how does PC know where the program starts? When entering a program under the direction of the assembler, there is no problem. It is simply a case of knowing the starting address of a program and quoting this in the manner laid down in the handbook of the particular assembler used. However, suppose there were no assembler and also suppose there were not even an operating system in ROM. In other words, suppose you were in the situation of having a perfectly serviceable piece of hardware but absolutely no software at all. Where would you start? As a matter of fact, the actual mechanism of loading the PC gives rise to a disturbing question which strikes at the root of stored program sequence control. This is the question – how was it possible to load PC with the starting address of the program unless there is *already* a program capable of performing the load action?

This is a chicken and egg situation because we can't fall back on the 'operating system'. The operating system is also a program so how was this loaded originally? Computer scientists have produced various solutions, although here we are concerned only with the one which is peculiar to the 6502 microprocessor family.

When the reset line (RES) is momentarily grounded (usually arranged to coincide with the closing of the power-on switch) the following series of events take place:

(1) All peripherals connected to the reset line are initialised to an orderly 'start-up' state. The interrupt mask is set to 1 to prevent the possibility of an interrupt during the start-up sequence.
(2) PC is loaded with the data which happens to be resident in the special addresses $FFFC (low-byte) and $FFFD (high-byte). The addresses were fixed during the design of the microprocessor.
(3) PC commences program execution because it now contains the starting address of the program.

From the above, it is evident that the writers of the operating system must ensure that the correct starting address is in $FFFC and $FFFD. It is equally evident that they must be in ROM (RAM can only be loaded with data by a program which already exists). Note that the concept of a vectored address allows the system programmer complete freedom to position the program anywhere. It would have been easier, of course, for microprocessor designers to lay down a mandatory starting address – say,

'all programs must start at address $0000'. This would allow PC to be initialised by a simple zero reset.

However, the vectored address approach is flexible and we should remember that infinite flexibility has always been the aim of computer scientists. There are three vectored addresses in the 6502 and, for completeness, these are shown in the following table.

6502 vectored addresses	
Vector address	Function
$FFFA and $FFFB	Non-maskable interrupt
$FFFC and $FFFD	Start-up/reset
$FFFE and $FFFF	Interrupt request

Although PC ensures that instructions are normally accessed and executed in consecutive order, there are times when the sequence must be broken. When a jump or conditional branch is encountered, the current contents of PC are altered drastically. In the case of an absolute jump, the entire contents of PC are replaced by the instruction operand. Branch instructions, however, use relative addressing rather than absolute. The operand is in the nature of an offset, which is added to, rather than replacing, the existing contents of PC. Since the offset is in two's complement binary (allowing positive or negative numbers) it is still possible to branch forward or backward.

The Instruction Register (IR)

The first byte of all machine code instructions is the operation code (abbreviated to 'op-code'). The code, which is different for every instruction and addressing mode, carries two vital pieces of information:

(1) What kind of operation is required.
(2) How many operand bytes (if any) are still required to complete the instruction.

After the code has been transferred from memory (known as the FETCH phase) it is routed via the highways to IR where it is held pending execution. If the decoding reveals that the instruction requires no further operand bytes (such as TXA,TAX etc) the instruction sequence enters the EXECUTE phase. If, on the other hand, decoding reveals that one or more operand bytes must follow, the sequence remains in the FETCH phase until the complete instruction has been received from memory.

The Data Register (DR)

The data bus carries information *downwards* from the microprocessor when writing to memory and *upwards* to the microprocessor when reading from memory. Because of this, DR operates as a bidirectional holding register, controlled by the R/W line. You will remember, from earlier discussions, that when R/W is in the high state (logic 1) the DR would be switched to the READ direction and when in the low state (logic 0), to the WRITE direction. The power levels on the raw bus are weak and *external buffers* are needed to boost the power. A full 64K of memory with additional peripheral loads could lead to degradation of logic levels, unless well buffered.

Whilst on the subject of the data bus, it is convenient to discuss the effect of *data jamming*. It is essential that all memory and peripheral devices connected directly to the data bus are equipped with 'tristate' outputs. That is to say, when the devices are in the disabled state, their connections to the data bus should be electrically impotent. Tristate devices ensure this by effectively *open-circuiting the outputs* during the disabled state.

The Address Register (ARL and ARH)

A 4-hex digit address describes a 16-bit logic pattern on the address wires A0 to A15. The address information can originate from several possible sources. It could originate from A, the output of the ALU or even the data bus. From whatever source, it will eventually be routed along the highway, ending up in the address register.

This register is split into two halves, each contributing a byte to the two-byte address. The lower order byte (A0 to A7) is held in ARL and the high order byte (A8 to A15) in ARH. As discussed earlier, the high byte determines the page address and the low byte the address on the page. The individual lines on the address bus are direct outputs of the registers. They are, of course, always outputs so the R/W control line is not involved.

It should also be noted that, unlike the data bus, devices connected to the address bus need not be tristate. This is because the address bus is always an output from the microprocessor intended to feed only the address decode circuits of memory or peripheral devices. Only the address registers can supply the bus so there is no possibility of data jamming by alternative logic voltage sources.

The microprogram

The term 'microprogram' has nothing to do with programs written for a microcomputer. In fact, microprograms are those which are buried inside the silicon of the microprocessor chip itself! It may surprise some readers that every instruction in the repertoire (about 200 in the 6502) requires its own special microprogram. A simple machine code instruction like LDA $72 is simple only from the viewpoint of the human intellect. In contrast,

logic circuits (which are baffled if required to answer any question with other than 'yes/no') require considerable assistance in dealing with LDA $72. They need micro-instructions, fed one at a time in order to open and close the appropriate register gates and activate the control lines. These micro-instructions must be given in the correct sequential order, for every individual instruction. Since a sequential set of instructions is, by definition, a program, then it is seen that the following statement is justified:

Every instruction has its own microprogram

It is not proposed to examine in detail each of these microprograms. This would take more space than this book allows. However, it is interesting to examine a possible microprogram for the simple instruction mentioned previously: LDA $72 will LoaD A with the data stored at address $72 on page 0 hex. This instruction consists of two bytes, which we will assume are residing at addresses $2E34,$2E35. The microprogram will first have to fetch these two bytes from memory in two phases of events – FETCH and EXECUTE.

(1) *The FETCH phase*
PC, having just dealt with the last byte of the preceding instruction, will already have been incremented to $2E34. A typical sequence would be:

(a) Contents of PC pass to ARL and ARH.
(b) The R/W line goes or remains high causing the op-code (LDA) to be read from memory and passed, via the data bus to DR.
(c) The contents of DR are then passed to IR and the instruction is decoded. From this decoding, the system now 'knows' there is a single operand byte to follow. PC is incremented to $2E35.
(d) The contents of PC pass to ARL and ARH.
(e) The memory is again read, causing the first operand byte ($72) to enter DR.
(f) PC is again incremented.

The complete instruction is now lodged in the microprocessor registers, ending the fetch phase.

(2) *The EXECUTE phase*
(a) The operand ($72) in DR is passed, via the highway, to ADL. ADH is cleared to zero (because it is a page 0 address).
(b) The memory is read, and the data at address $72 is passed to DR.
(c) The contents of DR are passed to A.

The instruction has now been executed with the PC left pointing to the address of the first byte of the next instruction.

The instruction chosen in the example was particularly simple and yet the microprogram was quite involved. It is left to the imagination to visualise the microprogram for ADC ($72),X (post-indexed indirect addressing).

Microprogrammers are a specialist breed and usually employed on the design staff of the chip manufacturer. It is fortunate that the brief outline above on microprogramming was included for interest only. The normal machine code programmer takes each complete instruction for granted and is oblivious to the existence of the internal microprogram steps. If we call machine code a 'low level language', then microprogramming is at ground zero.

The decoding matrix

Figure 2.2 showed the decode matrix. The function is to accept the op-code held in IR, decode it, and finally output a pattern of bits on the various gate and timing controls. This pattern will be different for every step in the microprogram. If this function is analysed carefully, we may come to the conclusion that the decode matrix will behave like a miniature computer with a number of fixed programs inside. We can relate IR to the 'program counter'. The op-code is only the starting address of the relevant microprogram. The 'words' read out from the ROM are the bit patterns supplying the various register gates and controls. These patterns will vary for each step of the microprogram. The gate controls are all hard-wired to the various registers. This wiring has been omitted from Fig. 2.2, however, to prevent an already complex diagram becoming incomprehensible.

Sub-pulses of the clock

It is not always appreciated that the clock pulses, which in the 6502 are running at 2 MHz (0.5μs period), are split up within the decoding matrix. Several sub-pulses are formed, each sub-pulse initiating each step of the microprograms. Within the matrix, the clock pulses are merely the 'low-frequency' envelope of the sub-pulses.

The Arithmetic and Logic Unit

Addition, subtraction and logical instructions will obviously be the responsibility of the ALU. However, in the interests of versatility, nearly all data is made to pass through the ALU irrespective of the particular instruction. For example, data can pass through the ALU without change by adding zero. This may seem time-wasting but is often justifiable. For example, address modification by indexing involves adding the contents of X or Y to the operand so the ALU is directly involved.

The 6502 is incapable of multiplication, division or exponential operations. It is not alone in this respect. It is very rare to find 8-bit microprocessors capable of performing any arithmetic instructions other than addition. Even subtraction is achieved by the roundabout way of adding the complement.

Before criticising these limitations, it should be remembered that the microprocessor was designed with the primary objective of controlling electrically operated devices and a primitive instruction repertoire was quite sufficient for the purpose. It was never intended to be the brain of a general purpose computer. However, if a machine can add, it is fairly easy to write subroutines which can multiply, divide and handle exponentials. Users of BASIC, or indeed most other high level languages, are unaware of the primitive capabilities of the microprocessor although they have to pay for it by reduced execution speed. Software solutions are always much slower than hardware implementations.

The new breed of 16-bit microprocessors are virtually second generation products many of which include instructions which perform direct multiplication and division at an impressive speed.

The design of an ALU is based on a parallel binary adder which can be considered as an arithmetical prototype. With this as a basic building block, it is a relatively simple exercise in logic to arrange gates for implementing exclusive-OR (EOR), logical anding (AND) and the inclusive-OR (ORA) functions. Finally, it would only require 'function select' inputs to complete the transformation. Four of these, driven by the output word from the control matrix, could activate any one of 16 functions.

Software Interrupt (BRK)

The details of all machine code instructions are given in the relevant chapters but it is convenient at this stage to introduce the BRK instruction. Interrupts are normally the prerogative of peripheral devices but BRK is software initiated. Superficially, it just stops the computer but a dig beneath the surface reveals some interesting side effects. The instruction takes 7 clock cycles to complete the following sequence:

(1) It sets the B flag in the process status register (P).
(2) Adds 2 to PC.
(3) Sets the I and B bit in P then pushes P on to the stack.
(4) Loads the contents of $FFFE into PCL and $FFFF into PCH.

The motive behind this seeming complexity is to aid the writing of software error traps during program development. It is common practice to put BRK at strategic 'bug-hazard' points. This would be useless if the sole function of break was to kill all program flow completely. However, it

will be seen from the above that a convenient loop-hole is prepared. Providing a routine is written, with the start address residing in the Break Vector at $FFFE/ $FFFF, control is automatically diverted to the routine rather than stopping dead. The routine must establish, by pulling P back from the stack, that the B bit was set as a result of a true BRK rather than as a genuine peripheral interrupt.

Summary

1. The Commodore uses the 6510A microprocessor which is virtually identical to the 6502.
2. All internal registers communicate by means of input/output gates at the entrance to the highways. The gate controls ensure that only data from one register output occupies a common highway.
3. Not all registers in the 6502 are directly programmable.
4. The single accumulator (A) is the only register equipped with arithmetic facilities. There are no incrementing or decrementing instructions for the accumulator.
5. The X and Y registers are used as transfer registers to and from A, loop counters and indexed addressing modes.
6. Address modification by indexing consists of adding X (or Y) to the operand address.
7. The process status register (P) is a collection of seven independent flag bits, each signalling some important result. Most instructions keep P updated. Conditional branch instructions depend on the state of these flags.
8. The N flag=1 if bit 7 was set. The Z bit=1 on a zero result. Neither is directly programmable.
9. The V bit=1 on a two's complement overflow result. It can be ignored if the data is unsigned binary. The V bit copies bit 6 during the BIT test. It can be directly programmed by CLV and SEV.
10. The B bit=1 if an interrupt occurs as a result of BRK.
11. The D bit can only be directly programmed. SED makes D=1 causing subsequent numerical data to be processed in BCD format. CLD makes D=0 causing subsequent numerical data to be processed in normal binary format.
12. SEI sets the I bit, preventing interrupt when requested by IRQ. It is cleared by CLI.
13. The C bit sets to 1 on detecting a carry out from the msb. The C bit can act as a ninth data bit.
14. The stack is any dedicated area in page 1 of RAM. PSH pushes A to stack. PLA pulls A back from stack.
15. The address of the next vacant stack address is maintained by SP.
16. SP is automatically incremented after a push and decremented before a pull. This causes the stack to rise and fall.

17. The contents of SP can be initialised or changed only by means of TXS.
18. Access to the stack must obey the rule – last in, first out (LIFO).
19. The stack is used by subroutines to hold the return address, pending RTS.
20. The program counter (PC) is the only 16-bit register and is not directly programmable. It is in supreme control of the program sequence by always pointing to the next byte in the program.
21. PC is altered directly by JMP. An offset may be added by relative address action during conditional branches.
22. A pulse on the reset line (RST) initialises peripheral devices to zero PC is then loaded by the contents of the address pointed to by the start-up vector at $FFFC,$FFFD.
23. The instruction register (IR) holds the op-code fetched from memory.
24. The op-code defines the instruction type and carries information on the number of operand bytes to follow.
25. The data register (DR) is a bidirectional buffer between the data bus and the highways.

Self test

2.1 What does the 'D' mean in DRAM chips?
2.2 Name one advantage and one disadvantage of static over dynamic ram chips.
2.3 Which 6502 register is 16 bits wide?
2.4 What does ALU stand for?
2.5 Describe two ways of incrementing the accumulator.
2.6 What is the maximum negative number which can be held in a single byte?
2.7 What is the hex contents of a byte which is holding the maximum positive number in two's complement form?
2.8 Under what circumstances will a programmer ignore the V bit in the process status register?
2.9 Assume the D bit in the process status register has been previously set to 1 and the accumulator contains 0001 1001. What will the accumulator contain after 0000 0010 has been added?
2.10 If the Z bit in the process status register is at 0, the last active instruction must have yielded a zero result. True or false?
2.11 Which bit in the process status register is often termed the ninth bit?
2.12 What do the initials LIFO mean?
2.13 The stack pointer can only refer to one particular page in memory. What is this page?
2.14 There are three fixed vector addresses in the 6502. Name them.
2.15 How is data jamming prevented in a paralleled bus system?

Chapter Three

The 6502/6510A Instructions and Addressing Modes

Initial terms and definitions

Some readers will be aware of the following points but it will be helpful to repeat them. In any case, the terms used to describe aspects of machine code are far from standardised. The complete instruction set is relegated to Appendix C which should be consulted frequently during reading this and subsequent chapters. When programming in a high level language such as BASIC, an individual order to the computer is called a *statement*. For example, E = M*C↑2 is an example of a statement.

In machine code, orders given to the computer are by means of *instructions*. Instructions are primitive and many are needed to form the familiar high level statements. An instruction will normally consist of an *op-code* to indicate the required action and an *operand* to indicate where the data is to be found. Sometimes, the location of the data will be obvious from the op-code but, in the general case, an operand is required.

There are several ways in which the operand can specify the location of the data. They are known as *addressing modes* and there are thirteen of them in the 6502 family although not all of these are available to every instruction. Because one byte is used for the op-code it would be possible to have 256 different ones. However, 90 of the possible combinations are reserved for 'future expansion' (illegal in other words). This leaves 166 valid instructions to choose from. The task of selecting the most suitable op-code is less bewildering than it appears from the figures. There are only 56 *completely different* instructions. It is the available addressing modes for each instruction which multiply the choice.

The op-codes are specified by means of a pair of hex digits. There is a different op-code for every variation of addressing mode. However, the hex coding is really of academic interest because machine code on the Commodore 64 should be entered by means of an assembler. The details of an assembler will be discussed fully in Chapter 4. The most useful property of an assembler is the facility to enter op-codes in three-letter *mnemonic* form. The desired addressing mode is indicated by the form in which the operand is written. The repertoire of instructions is set out formally in

Appendix C. Consequently, the purpose of this chapter will be to explain the symbols, to define the addressing modes and to offer guidelines on the choice of a particular instruction and the most suitable method of addressing.

Factors influencing choice

It is not easy to give a specific answer to the question 'What is the correct instruction to use here?' The choice is very often a compromise between execution speed, memory economy and the demands of structure. Newcomers to machine code may be quite satisfied if their subroutine works at all but it soon becomes apparent that there are good and not so good variants. It is popularly supposed that a program written in machine code will always be much faster and take less memory than the BASIC version. This is a reasonable generalisation but not a universal truth. A poorly written machine code program could be slower than the BASIC equivalent. Even if it is faster, it is well to remember that a speed advantage, to have any real meaning, must be assessed on human, rather than machine, time scales. If a BASIC version runs in one second and the machine code version runs in a millisecond, the advantage is academic rather than visible. The items of information needed to assess the merits of each instruction are as follows:

(1) What does it do? This information is conveyed by a three-letter mnemonic such as LDA or ADC. Although the mnemonic itself conveys a reasonable idea of what the instruction does, it is primarily intended as an aid to the interpretation of a listing. It cannot cover all the subtleties. It is necessary to augment the mnemonic by either a verbal definition or a loosely standardised format known as *operational symbols* (discussed later).
(2) What addressing modes are available?
(3) What flags in the process status register are altered (updated)? Ignorance or confusion in this area is the cause of many an intractable bug.
(4) How many clock cycles does it consume? The number of clock cycles is influenced more by the addressing mode than the actual instruction. Clock cycle time is particularly critical if the instruction is within a loop which revolves many times. Outside a loop, it is seldom important enough to influence choice.
(5) How many bytes are in the instruction? All instructions take at least one byte because they all have an op-code. The operand, however, can be absent altogether, one byte long, or two bytes long. Knowledge of the number of bytes required can be helpful. For example, it can be a matter of doubt in certain circumstances whether to write $004B or $4B in the operand. They are mathematically the same but an incorrect choice can cause havoc to the program.

(6) What is the hex op-code? Programming will always be performed with the aid of the assembler which uses mnemonic op-codes. However, it is still necessary at times to be aware of the hex coding for every instruction because the assembled machine code program will include it. It is easy to use an incorrect address mode by mistake when writing the operand but the hex code, which is specific for the addressing mode, might highlight the error during debugging. It is interesting, but not particularly rewarding, to write out the hex code in binary. It gives an insight into the mind of the microprocessor designer because some intriguing patterns emerge which can give a clue to the microprogram within the chip.
(7) What is the correct syntax for the operand? This depends on the addressing mode and the rules are rigid, more so than in BASIC. The assembler does its best but it would be foolish to add user-friendliness to its list of virtues. Make a mistake and you are on your own!

Operational symbols

Universities have traditionally considered computing and data processing subjects to be the prerogative of the mathematics department. The computer is useful as a tool in mathematics so it was considered natural that computing should be taught by mathematicians. Whether this has helped or hindered progress may be arguable. There is no denying that a mathematical brain was behind the establishment of operational symbols.

How do we describe exactly what an instruction will do, bearing in mind that there must be one, and only one, interpretation? Normal language is one way; perhaps the obvious way. But, to a mathematician, normal language lacks precision and is difficult to formulate concisely without using a lot of ifs and buts. Operational symbols are concise and unequivocal. They explain what the instruction does but make no attempt to explain the meaning of the operand. This is understandable because the meaning of an operand depends only on the addressing mode chosen. For example, the instruction LDA will have the same operation symbols whether it is using immediate, zero-page, absolute, indexed or indirect addressing. The general pattern of operational symbols is of the form:

Action → Result

The arrow denotes the direction of data transfer and is preferable to the = sign sometimes used. The abbreviations used for the registers are those already used but M is used to represent the data specified by the operand. As a simple example, the instruction STA could be described as follows:

A → M

This means 'Store a copy of the contents of the accumulator in the address specified by the operand'. Note that the arrow *points from the*

source to the destination and only the destination contents are over-written by the new data; the source data is preserved.

To take a little more complex example, the instruction ADC could be described concisely as follows:

$$A+M+C \rightarrow A$$

This means 'Add together the present contents of the accumulator, the data specified by the operand, and the carry bit, then place the result in the accumulator'.

The shift and rotate instructions are fearsome looking. For example, the instruction ASL (which is Arithmetic Shift Left) has the operational symbolism:

$$C \leftarrow (7...0) \leftarrow 0$$

The bracketed expression indicates the bits within a byte numbered 0 to 7. The action shows that a zero enters from the right and overspill from bit 7 goes into the carry.

Classification of instructions

There are many ways of classifying instructions. Appendix C simply lists them in alphabetical order by mnemonic group. This is useful as a quick reference but is by no means a scientific classification. Appendix C2 classifies them according to the flags affected in the processor status register and can be quite useful. Appendix C4 is an attempt to classify them according to 'popularity'. It is undeniable that some instructions out of the 56 are used a lot, some are used at times and a few are used spasmodically. Unfortunately, the choice of instructions to perform a given task is very much an individual affair. Some programmers have a particular liking for a certain subset. Indeed, it is often possible to recognise a friend's handiwork from the listing which can be almost a fingerprint. Because of the individual character, Appendix C4 can be no more than the author's personal choice although it might help those who are initially bewildered.

In this chapter, the instructions will be introduced (rather than classified), according to need. No account will yet be taken of the various addressing modes under each mnemonic.

Finding temporary homes for data

Due to the single accumulator in the 6502, it is often necessary to find a temporary home for existing data. There are several choices:

(1) Transfer A to another register by the use of TAX or TAY and later restore by TXA or TYA. This is the simple and speedy solution because

they are both single-byte instructions, taking only two clock cycles. The trouble is that existing data in the X and Y registers may also be valuable and must not be overwritten. X and Y are often totally committed for indexing or loop counting.
(2) Push A to stack by using PHA and retrieve later by PLA. These are single-byte instructions but they take three clock cycles. It is important to bear in mind the LIFO (last in first out) nature of the stack. Mistakes in the order of retrieval could result in false data entering A. Another danger, of course, is stack overflow although this should be a comparatively rare event.
(3) Store A in a memory location by use of STA and retrieve it with LDA. This will take three clock cycles if the location is on page zero and four on any other page (indexing and indirect addressing can take five or six cycles).

Performing arithmetic

There are only two direct arithmetical instructions, ADC and SBC for addition and subtraction respectively. The carry is always involved and, to avoid introducing garbage carries left over from a previous operation, it is important to be aware of the following rules:

(1) Before using ADC, the carry should normally be *cleared* with CLC.
(2) Before using SBC, the carry should normally be *set* with SEC.

Although in some circumstances the carry can be treated as the 'ninth bit', it should be borne in mind that this is purely a way of looking at it. Obviously, this ninth bit is not transferred by STA, TAX or TAY.

Addition and subtraction of single byte numbers are, of course, severely limited in the range of the result (255 in unsigned binary and +127 and −128 in two's complement binary). Fortunately, the carry bit allows double or multiple byte numbers to be added or subtracted because it can act as the continuity element between the msb of one byte and the lsb of the next. Thus, the carry is only cleared before the two lower order bytes are added. The higher order byte additions will include the carry over (if any) from any previous process so it would be fatal to clear the carry first.

It is important not to forget that there are two arithmetic modes depending on the D flag being set or cleared. The default condition is D = 0, which is the normal two's complement binary arithmetic mode. It is wise, though, to ensure the default condition by initialising with CLD at the head of a program. On the rare occasions when decimal (BCD) mode is required then the initialisation begins with SED, but remember this mode *continues until cancelled again*.

Multiplication and division is possible by a tongue-in-the-cheek method using ASL and LSR respectively. The operations are limited to integral

powers of two. Watch must be kept on overspill from the msb in multiplication and the lsb in division.

Subject to overspill into the carry, shifting left by ASL will multiply by two each time so four consecutive ASL operations will multiply the existing data by 16. Division by two is achieved by LSR although we must remember that the overspill from the right (from the lsb) goes into the carry. As a matter of interest, the reason why LSR is named Logical Shift Right is due to this very reason. It is arithmetically absurd for carry status to be in the lsb position, hence it is deemed to be 'logical' shift. This is in contrast to ASL (Arithmetic Shift Left) where the carry action is at the msb end. Unless the programmer is sure, from previous knowledge of the data, multiplication and division by these instructions must check for the presence of a carry after each use.There will be exceptions, of course, such as when multiple-byte precision is used. In these circumstances, the carry will be providing continuity between the component bytes when used in conjunction with ROL or ROR.

Clearing memory and registers

There are no instructions in the 6502 which can clear any of the registers or memory locations to zero. The usual way to clear registers is to store zero in them. To clear memory locations, a previously zeroed register can be stored in them. Those who are fascinated by novelty may be attracted by the following little snippet:

> Exclusive-oring data with itself always results in all zeros.

For example, if A contains $9D and we write EOR #$9D, the accumulator result is $00. (To confirm, write out the example in binary form.)

Up-counting and down-counting

Counting is essentially an adding-by-one operation and implies 'up-counting'. It is also called *incrementing*. Down-counting is subtracting by one. It is also called *decrementing*. The X and Y registers can be counted up or down by the single byte instructions INX, INY, DEX and DEY, each taking only two clock cycles. Data in memory can be incremented or decremented by means of INC or DEC but not economically. They each take five to seven cycles depending on the addressing mode in use.

The accumulator is left out in the cold, lacking an increment or decrement instruction. It can, of course, be done by adding or subtracting 1 which, like DEX or INX only takes two clock cycles, but it requires two

bytes even for the immediate addressing mode. There is also the possibility that the carry might have to be cleared first which, if forgotten, could lead to a mystery bug. An alternative is some roundabout method such as TAX then INX then TXA, providing of course, that X (or Y) is free.

Counting is an essential part of loop control. The number of loop revs can be achieved either by starting with N and *counting down to zero* or starting with 1 and *counting up to N*. The advantage of the count down method is that testing for loop exit can be achieved with BNE or BPL. Unfortunately, it is very easy to be 'one out' in the count down. If we count up to N, an extra comparison instruction such as CPX, CPY or CMP is required to check the exit condition but the method may have the advantage of seeming more 'natural' and errors by one are less likely.

Processing particular bits

There will be times when it will be required to operate on one or more particular bits within a byte, rather than on the entire byte. We may wish to ensure, say, that bit 3 is set to 1 without altering the remaining bits. The possible operations fall into three main groups, *clearing* bits to zero, *setting* bits to 1 and finally, *changing* bits. This is achieved by using one of the three 'logical' instructions AND, ORA and EOR in conjunction with the appropriate mask word in the operand. The action is always on the accumulator.

To clear selected bits:
Use AND with an operand mask as follows: '1's in the mask will leave corresponding bits unchanged. '0's in the mask will ensure that corresponding bits are 0.

To set selected bits:
Use ORA with an operand mask as follows: '0's in the mask will leave corresponding bits unchanged. '1's in the mask will ensure that corresponding bits are 1.

To change selected bits:
Use EOR with operand mask as follows: '0's in the mask will leave corresponding bits unchanged. '1's in the mask will ensure that corresponding bits are changed.

The following examples may help in understanding how to work out the correct mask:

(a) To ensure that bit 5 in the accumulator is a 0, use AND #$DF (the mask in binary is 1101 1111).
(b) To ensure that bits 2 and 6 in the accumulator are '1's, use ORA #$44 (the mask in binary is 0100 0100).
(c) To ensure that bit 3 in the accumulator is changed, use EOR #$08 (the mask in binary is 0000 1000).

One's complement of accumulator
It is sometimes necessary to *flip* all the bits in a byte (i.e. produce the one's complement). Assuming the data is already in the accumulator, this can be done by exclusive-oring as follows:

```
EOR #$FF or EOR #255
```

Two's complement of accumulator
The two's complement is obtained by adding 1 to the above. Unfortunately, we can't add the 1 by incrementing because the result is in the accumulator. The only way is to follow with ADC #1, making sure to clear the carry first. The coding is as follows:

```
EOR #$FF
CLC
ADC #1
```

Since the two's complement of X is 0−X, an alternative method is simply to subtract the number from zero. This is, by definition, the two's complement but would entail storing the data first before loading the accumulator with 0.

Finding the state of a particular bit
It is sometimes important, particularly in peripheral control, to find out the state of one particular bit within a byte. This can be done by loading the byte into the accumulator, erasing all bits except the one of interest, then testing for zero. If the result is non-zero, the bit must have been a 1. For example, suppose we are interested in bit 3, the coding could be:

```
LDA data
AND #08 (0000 1000)
BNE etc.
```

An alternative method, which only works if bit 6 or bit 7 is involved, is the BIT test. For example, we can start by writing:

```
BIT data        ('data' is an arbitrary address)
```

This copies bit 6 and bit 7 of the data into the V and N bits respectively. This can be followed by BVS or BMI as required. The BIT instruction takes 3 clock cycles if data is on page zero but otherwise 4 cycles. As a bonus, the bit test also logically ANDs the data into the accumulator. If this is a nuisance rather than a bonus, the accumulator should be stored first. Because of this, use of the BIT test is not a commonly used instruction.

Besides the three logical instructions AND, ORA and EOR, the shift and rotate instructions LSR, ASL, ROR and ROL are also used to play around with bits. LSR and ASL should be thought of as 'open-loop' operations because bits are lost if the carry is already full. In contrast, ROR and ROL are 'closed-loop' because the bit pattern circulates. They can all play an important role in peripheral work and some off-beat requirements. The shift

and rotate instructions are unique in having 'accumulator' addressing. Thus, they can act on the accumulator or a memory location. If the action is required on the accumulator, the mnemonic must be followed by A. For example, to shift the accumulator right, we must write LSR A. When using accumulator addressing, no operand is necessary (the 'A' is not a true operand and does not consume a byte). Because of this, it should be noted (because it is a common mistake) that the shift and rotate instructions must either have an operand or an 'A'. For example, a naked LSR is illegal.

Double-byte multiplication

This provides a useful exercise in shift and rotate operations. Although ASL and ROL both multiply by two, the carry can be a problem if they are not chosen wisely. No carry must be allowed to enter the lower order byte from the right so ASL is appropriate. On the other hand, the higher order byte must take into consideration the carry from the right so ROL must be used. Assuming the data is in two bytes of memory, the coding would be:

```
ASL low-byte
ROL high-byte
```

Double-byte division

The opposite is required here. Thus, the higher order byte must be attacked first and a carry must not be allowed to enter from the left. This suggests LSR as the first step. The lower order byte must receive a carry (if any) from the left so the correct instruction here is ROR. Assuming that the data is in two bytes of memory, the coding is therefore:

```
LSR  high-byte
ROR  low-byte
```

Branching techniques

The equivalent of the dreaded GOTO in BASIC is JMP. The jump to a new part of the program is unconditional and, because JMP has a two-byte operand, can reach any part of the 64K memory map. Appendix C lists seven conditional branch instructions. A common cause of a programming bug is an incorrectly used branch test allowing an unexpected loophole. The following points are worth emphasising:

(1) Branch instructions themselves have *no effect* on the processor status register. Thus, two different branch instructions can follow one another so the original data can be tested for two conditions.
(2) BMI or BPL should only be used if data is represented in two's complement binary. They are meaningless in unsigned binary because there is no differentiation into positive or negative sets.

(3) Before using a branch, make certain that the *last operation* actually updates the bits you are testing. In other words, check up on Appendix C2 which includes a classification of all instructions according to their effect on the processor flag bits. For example, it may be pointless to use BCC after DEX because only the N and Z flag bits are updated.

The limits of +127 byte forward or −128 bytes backwards has been covered elsewhere. If the branch is beyond range (which should not be often) the customary solution is to combine the branch with a JMP. For example, suppose the branch is to be BNE LOOP and the label 'LOOP' is out of range. The conventional way out is as follows:

```
BEQ SKIP
JMP LOOP
SKIP
```

Note that the opposite test (BEQ) is used instead of BNE so the jump is leap-frogged to the label SKIP.

Comparisons

It is often required to compare two numbers in order to set the status flags without altering the contents of the register. There are three instructions which perform this task, all of which set the N, Z and C flags:

CMP, which compares memory with the contents of the accumulator.
CPX, which compares memory with the contents of the X register.
CPY, which compares memory with the contents of the Y register.

The comparisons are done by subtracting the memory data from a copy of the register in question. The operational symbolism is therefore A−M, X−M or Y−M respectively. It is easy to get mixed up with the direction of the subtraction, so note carefully that the subtraction is from the register. A suitable branch instruction must follow a comparison (otherwise there would be no point in asking for the comparison). It is possible to get in some funny mix-ups. The following examples may help in choosing the correct branch:

(1) To check if the register is *less* than memory, follow with BCC.
(2) To check if the register is *equal* to memory, follow with BEQ.
(3) To check if the register is *greater* than memory, follow with BEQ first then BCS.
(4) To check if the register is *greater than or equal to* memory, follow with BCS.

Addressing modes

Commencing with a definition, an addressing mode is the significance to be attached to the operand part of the instruction. Addressing modes available on the 6502 can be conveniently divided into three groups: non-indexed, simple indexed, and indirect indexed. Most of these modes may already be familiar to most readers. However, some revision or restatements are advisable, if only to maintain continuity during the lead-up to the rather nasty (nasty to grasp, that is) indirect addressing modes. Appendix C3 classifies instructions according to the addressing modes available.

Implied addressing

This is the simplest addressing mode in the repertoire because memory is not involved, neither is an operand required. They are all single byte instructions, conveying full information by the op-code alone. They all refer to internal operations on the 6502 registers. Because most of them only take two clock cycles, they are, or should be, the popular choice wherever possible.

Instructions which allow implied addressing and consume only two clock cycles are: CLC, CLD, CLI, CLV, DEX, DEY, INX, INY, NOP, SEC, SED, SEI, TAX, TAY, TSX, TXA and TXS.

The following take more than two clock cycles: BRK, PHA, PHP, PLA, PLP, RTI, and RTS.

Immediate addressing

Memory is not involved because the operand is the data. All instructions using immediate addressing consume two bytes: one for the op-code and one for the operand. The standard assembler prefix to denote this mode is the symbol (#). For example:

```
LDA #32 or LDA #$20
```

Both are using immediate addressing. The first example is loading the decimal number 32 into the accumulator while the second example loads hex 20. Whether to use hex or decimal is optional but the guiding rule is to choose the more natural form for the purpose in use. For normal numerical work, decimal would be the preferred notation but for AND, EOR or ORA masks, hexadecimal has more meaning. Although it may seem to be stating the obvious, the largest numerical operand is 255 or $FF because immediate addressing only allows a single byte operand.

Immediate addressing is used for constants, particularly in conjunction with comparison instructions at the end of a loop as, for example, CMP #20. The constant must, of course, be known to the programmer at the time of writing. In BASIC, we are usually extolled to avoid constants within the body of the program, the advice being to assign them to a variable at the

head of the program. Such advice is not necessarily sound when applied to machine code because this would mean a trip to memory to obtain the data. The power of immediate addressing lies in the fact that memory is not involved: the data is *immediately* available in the instruction, providing, as said before, the programmer knows it at the time of writing.

There are eleven instructions which allow immediate addressing: ADC, AND, CMP, CPX, CPY, EOR, LDA, LDX, LDY, ORA and SBC.

Absolute addressing

We should begin by sorting out some of the confusing terms used by different authorities. The term 'direct' addressing is often used loosely when the operand refers to the *address* of data, rather than the data itself. Thus, the instruction LDA $0034 is an example of 'direct' addressing (note there is no '#' prefix). The instruction causes the contents of address $0034 to be placed in the accumulator. However, bearing in mind that 6502 has a 64K memory map, it will be evident that addresses between $0000 and $00FF would result in an inefficient use of memory space if the full four-hex digit address were mandatory. Since the data bus is only eight bits wide, the microprocessor would need to make two trips down the address and up the data bus to collect the full operand. The first two leading zeros are useless passengers.

To improve the efficiency, the address space is broken down into two domains. As mentioned in an earlier chapter, addresses within the range $0000 to $00FF are designated the page zero domain, to distinguish them from all other addresses $0100 to $FFFF. With regard to the terms used, the Motorola 6800 (the ancestor of the 6502) used the term 'direct' addressing instead of zero-page addressing and 'extended' addressing to cover the rest. Many machine code programmers, brought up on the 6800, had to readjust to the change in terminology. Returning to the 6502, the term 'absolute' addressing is applied to addresses anywhere in the 64K memory map. In other words, absolute addressing requires four hex digits, while zero-page addressing only requires two. Instructions using absolute addressing require three bytes, one for the op-code and two for the operand.

There are 21 instructions which allow absolute addressing. These are: ADC, AND, ASL, BIT, CMP, CPX, CPY, DEC, EOR, INC, JMP, JSR, LDA, LDX, LDY, LSR, ORA, ROL, STA, STX and STY.

Zero-page addressing

The concept of zero-page (sometimes called page-zero) is so important that it justifies emphasising the boundaries once again.

> Zero-page is the address range $00 to $FF or 0 to 255

There are reasons why this page deserves special treatment. There are obvious speed advantages, due to the single byte operand. This also leads to a saving in program memory space. Another reason is that the more complex addressing modes (to be dealt with later) require address pointers which must be in zero-page. Perhaps the most disappointing aspect is the scarcity of available space. The operating system, not surprisingly, utilises the vast majority of zero-page.

Free space in zero-page is between $FB and $FF inclusive

Because of the restricted space, it is essential, before planning any ambitious machine code systems, to choose zero-page locations with care. The apparent speed advantage is not, in itself, sufficient to justify squandering locations. In fact, it is sound philosophy to treat zero-page locations in the same light as registers – as valuable and scarce commodities. A good rule is to use zero-page for the most frequently used variable data. Sometimes, it may be wise to use zero-page for data within a loop, even if it means temporarily transferring it from an absolute address and then back again. The advantage of this approach may be appreciated more readily if we examine a few figures. Suppose a variable data item, located in an absolute address, is in the middle of a long loop which revolves 10000 times. Suppose we then transfer it temporarily to zero-page before entering the loop by using:

```
LDA $xxxx      (absolute, 4 clock cycles)
STA $xx        (zero-page, 3 clock cycles)
```

After the loop ends, the status quo can be regained with:

```
LDA $xx        (zero-page, 3 clock cycles)
STA $xxxx      (absolute, 4 clock cycles)
```

The four extra instructions for the complete transfer have taken a total of 14 clock cycles and consumed an extra 10 bytes of programming space. The saving within the loop, however, would be 1 cycle per rev, leading to a total saving of 10000−14 = 9986 clock cycles. We shall see later that indirect address pointers in zero-page will take two bytes each and many of these may be required in a program of even moderate complexity.

Relative addressing

Relative addressing is only used with *branch* instructions. In fact, forgetting the assembler for a moment, relative addressing is the only method possible in branch instructions. Using hex machine code as an example,

```
BEQ $04
```

The literal meaning is 'If equal to zero, branch 4 bytes forward'. The term 'relative' refers to the program counter. If the branch conditions are satisfied, the program counter (which always contains the address of the next program byte) has 04 added to it. This causes the next byte executed to be 04 bytes ahead, relative to the previous position. In other words, the operand indicates the number of bytes to be skipped. To branch backward, the two's complement is required (see Appendix A) so, to branch 04 bytes back, the instruction would be BEQ $FC. Clearly, the calculation of the correct operand is an error-prone exercise. The assembler takes all the drudgery out of relative addressing by allowing the operand to be a label instead of a relative address. We can use,

```
BEQ Loop
```

This works, subject to the proviso that the line, to which we wish to branch, is prefixed with the 'Loop' label (naturally, the choice of label is arbitrary). The assembler is hiding from us the fact that relative addressing is being used. Instead, it appears as a simple 'branch to label' operation which is far less error-prone than grappling with relative addressing.

As for the timings of relative addressing, these will depend on whether or not the branch is taken. If taken, a branch takes 3 clock cycles or, if across a page boundary, 4 clock cycles. If it is not taken, the branch takes 2 clock cycles.

The extra cycle when a page boundary is crossed is due to the alteration to the *high-* as well as low-byte of the addresses. If speed is very critical, a programmer should watch the hexadecimal assembly listing closely to see if a page boundary is crossed. For example, suppose the program counter was showing $05FC prior to a branch. If the relative branch is $04 ahead, the new program counter reading would be $0600, therefore there has been a boundary crossing between page 5 to page 6 which consumes an extra clock cycle. If such a branch was in the middle of a loop which revolves N times, it would be sensible to manipulate the coding, or alternatively relocate, so that the branch range was limited to the same page, and saving N clock cycles. It is surprising how attention to such small details can result in a material gain in execution speed. Although terribly wasteful in terms of memory, it is better to cut loops out altogether and resort to straight in-line coding if speed is absolutely vital. In most cases, this will be little more than an idealistic solution.

Indexed addressing

Although briefly discussed elsewhere, the concept of indexed addressing deserves detailed treatment. The indexing mode is denoted by a *comma* following the operand, followed in turn by X or Y. For example:

```
LDA $2356,X or LDA $75,Y
```

Both are examples of indexed addressing but the first is using an absolute

address and the second is using a zero-page address. The contents of the X (or Y) register is automatically added to the operand address before the instruction operates on the resultant address. It is well to recap on the terms used in indexed addressing:

(a) The base address is the address as stated in the operand.
(b) The relative address is the contents of index register (X or Y).
(c) The absolute address is the sum of the base and relative addresses.

As an example, assume that X contains 3 and that the instruction LDA $34,X is written. The *base* address is 34, the *relative* address is 3 and the *absolute* address is 34+3=37. Alternatively, the term 'effective' is often used in place of 'absolute'.

Two forms of indexed addressing are recognised:

(1) *Absolute indexed*, when the operand is any address in the 64K memory map. The instructions allowing X as the index register are ADC, AND, ASL, CMP, DEC, EOR, INC, LDA, LDY, LSR, ORA, ROL, SBC, and STA. The Y index register can be used in ADC, AND, CMP, EOR, LDA, LDX, ORA, SBC and STA.
(2) *Zero-page indexed*, when the operand is on page-zero. The instructions which allow X as the index register are ADC, AND, ASL, CMP, DEC, EOR, INC, LDA, LDY, LSR, ORA, ROL, SBC, STA and STY. There are only two instructions which allow the Y register for indexing. They are LDX and STX.

A mysterious bug can occur when using zero-page indexed addressing if the contents of X plus the operand address come to more than 255 or $FF. Clearly the single byte operand cannot hold numbers of this value so a wrap-around takes place. For example, if the instruction is LDA $FE,X and X contains 2, the arithmetical sum would be $100. The wrap-around action, however, will mean that the first hex digit is dropped and the absolute address will be $00 instead of $100.

Indexing allows any item in a block of data to be addressed by suitable adjustment of the index register. The operand of an indexed instruction (the base address) can be the address of the first item in the block or the last, depending on convenience or the programmer's whim. For example, if the base address is to be the start of the block, the index register can be incremented (by INX) within the loop until the last item is reached. On the other hand, it may be more convenient to choose the end of the data as the base address, in which case the index register is decremented (by DEX) until the first item is reached. Decrementation of the index register towards zero is generally recognised to be the more efficient method because the end-of-loop test can be carried out by a simple branch, such as BNE. The incrementation method demands a comparison (CPX or CPY) before the branch test. However, program legibility is sometimes more important than speed. There is a natural incliniation to count up towards a finite limit

rather than to count down towards zero and there is less chance of being 1 out in the count.

Besides accessing a data block sequentially, indexing is useful for *look-up tables*. For example, imagine a table of sines (or other mathematical functions) between, say, 0 and 89 degrees to be stored in a data block and that the base address is where sin(zero) is located. The table can be accessed by

```
LDA base, X
```

If the required angle is in X, the sine of the angle will be in the accumulator. The limitation of 8 bits for each sine will only give an accuracy to about two decimal places unless multi-byte working is used. Also, the programmer must take account of the decimal point when interpreting the result. Obviously, it would be absurd to use this method in place of the resident BASIC trig functions unless high speed access is vital.

Indexing is really address modification made easy. Besides being interesting, it is worth examining an alternative method (which was, historically, used before index registers were thought of) involving direct modification of the operand. This consists of loading the operand of an instruction into the accumulator (or other register), changing its value and then returning it to the previous location. To see how this works, consider the following line:

```
STA blogs
```

The operand has an arbitrary symbolic address. If this were in a loop and we wished to store the next item in blogs+1 without using indexing, it could be achieved as follows:

```
Modify STA blogs
       INC Modify+1
```

Note that the original line has now been given an arbitrary label 'Modify' which is where the op-code STA is stored, so blogs must be located in the next address, Modify+1. The next line increments the contents of blogs+1 so we have achieved 'address modification' by a roundabout method. If the change is to be more than just a simple increment – say, adding 7 – the coding could be as follows:

```
Modify STA  blogs
       LDA  Modify+1
       ADC  #7
       STA  Modify+1
```

Such direct alteration of an operand by the program itself is sometimes useful, but it is not a practice to be recommended. Listings of machine code are never easy to follow and these sorts of tricks can only add to the general confusion. It is worth emphasising that the primary function of an index

register lies in the ability to alter the *effect* of an operand and without *altering the operand* itself. One disadvantage of the 6502, which soon becomes evident in the early stages of programming, is the limit of 8 bits. This, of course, restricts the range of addresses which can be scanned by indexing even when absolute indexing is used.

Indirect addressing

Mastering any subject consists of systematically overcoming the various intellectual hurdles which appear during a course of study. Student drop-outs may occur when a hurdle is reached which is just too high. In machine code programming, there are many hurdles to overcome but the one which is responsible for the greatest student drop-out ratio is the concept of *indirect addressing*. Indexing is relatively easy to grasp once the advantages of address modification are realised but the following definition may help in understanding why difficulties arise in indirect addressing.

> An indirect address is the address of an address.

In assembly language, indirect addressing is indicated by enclosing the operand in parentheses as follows:

```
LDA (operand)
```

Note that although the operand is indeed an address, it is where the computer must go to find the address of the data. We shall continue for the moment to use LDA in examples, but it should be mentioned that simple indirect addressing as described above is only available with one instruction, JMP. Providing this is borne in mind, there is no harm in continuing with LDA in the initial stages. Consider the instruction:

```
LDA ($70)
```

Because of the parentheses, $70 is an indirect address, referring the computer to go to a *double-byte* address $70 (low-byte) and $71 (high-byte). This double-byte address is known as the *address pointer* because it 'points' to where the required data is located. Continuing with the examples, suppose that address $70 contains $35 and address $71 contains $0D. Returning now to the original instruction, LDA ($70), it should now be apparent that the contents of address $0D35 will be loaded into the accumulator. We will further assume that $0D35 contains $56.

Let us recap, using this example to illustrate the terms once more:

The instruction was LDA ($70).
The indirect address is $70.
The address pointer is $0D35.
The data pointed to and finally loaded into the accumulator is $56.

Figure 3.1 may help in the understanding of the above example.

When first introduced to the idea of indirect addressing, it is difficult to grasp the use of it. It appears to be a complicated and tortuous path to follow, merely to place data in the accumulator. For instance, it is

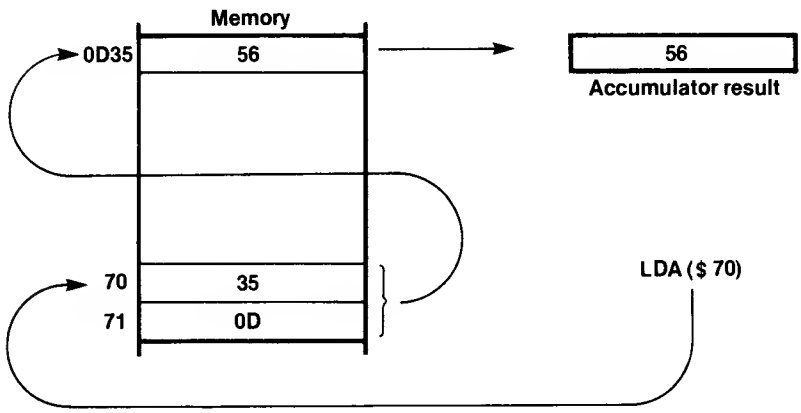


Fig. 3.1. Data flow in indirect addressing.

understandable, and pertinent, to ask why the line in the above example couldn't have been written in the simpler absolute addressing form:

```
LDA $0D35
```

After all, it may be argued, both forms would have identical effects. They would both load the same item of data into the accumulator, but the second form would not be wasting a valuable location ($70) in zero-page and would certainly be quicker to execute. The answer to this lies in the ability of indirect addressing to alter the effect of an operand without altering the operand itself. You will remember that this quality was the fundamental justification for the use of indexed addressing. If the address pointer is changed in any instruction using indirect addressing, the effect of the instruction acts on a different location. This has far-reaching advantages, particularly when writing general purpose machine code subroutines. Clearly, when writing a subroutine intended to act on a block of data, it would be restrictive to force the writer of the program using the subroutine always to place the data in a fixed memory block. However, with indirect addressing, all that is necessary is for the main program to know where the address pointer is ($70 and $71 in our previous example) and load it with the starting address of the data block. This flexibility means that the writer of the machine code subroutine need have no knowledge of the whereabouts of the eventual data block.

Before proceeding further, it should be remembered that the descriptions so far have been simplified by assuming that a 6502 has the instruction LDA (operand). Apart from the single instruction, JMP, simple indirect

addressing is not supported. Instead, we have the added benefit (and unfortunately, the added complication) of indirect addressing combined with indexing. In fact, there are two forms to choose from, called 'indirect indexed' and 'indexed indirect'.

Indirect indexed addressing

This is the form most often required. Only the Y index register is allowed in this mode. The assembler form is:

```
LDA (operand), Y
```

The operand is single byte and therefore can only refer to a zero-page address.

The only difference between this mode and simple indirect addressing is the addition of the Y register contents to the address pointer. That is to say, the operand still defines where a double-byte pointer is located but the pointer is modified by the *addition of the Y register contents*. As an example, assume that the following line is written:

```
LDA ($70), Y
```

Also assume that the contents of address $70 contains $35, address $71 contains $0D and the Y register contains $02. The effective address pointer will be $0D35 + $02 = $0D37. The effect of the instruction is therefore to load the contents of address $0D37 into the accumulator. The example figures can be used to define a few more terms connected with indirect indexing:

The instruction was LDA ($70), Y.
The base address pointer was $0D35.
The relative offset in Y was $02.
The effective address pointer was $0D37.

Figure 3.2 illustrates the example.

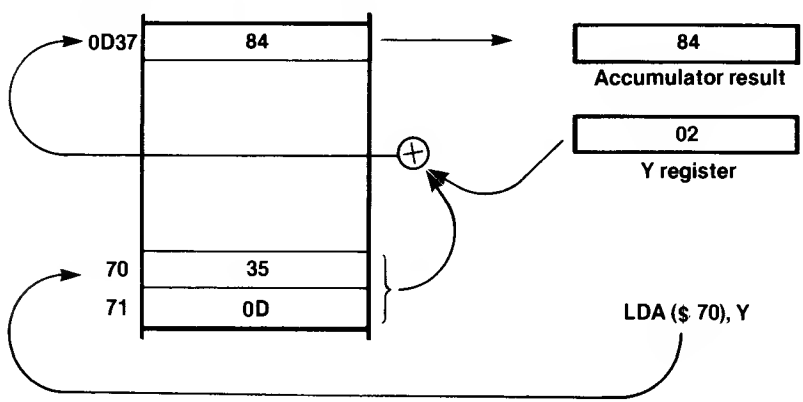


Fig. 3.2. Data flow in indirect indexed addressing.

Indirect indexed addressing allows the effect of the operand to be altered in either of two ways, by changing the base address pointer, by altering the contents of the Y register or both. The index register should be looked upon as an optional extra because there is no need to use it actively. For example, if Y is reset to $00, the instruction,

```
LDA ($70), Y
```

has the same effect as the simple (but fictitious) indirect addressing example given earlier:

```
LDA ($70)
```

However, an obvious use of indirect indexing lies in *sequencing through a block of data items* by incrementing or decrementing the Y register. It is helpful to distinguish simple indexed loops from indirect indexed loops by considering under what circumstances they would be used:

(a) Use simple indexing if the base address is known and constant.
(b) Use indirect indexing if the base address is not known at the time of writing or is liable to require changing.

One advantage of indirect addressing not yet mentioned is the ability to reach any part of the 64K memory map by use of a *single-byte* operand. This is because the address pointer in zero-page is double-byte (16 bits).

The following example is outline coding to perform a process on a block of memory with just sufficient detail to illustrate indirect indexed addressing. Assume that the address of the first data item has been prior assigned to the address pointer in $70 (low-byte) and $71 (high-byte) and the length of the block minus 1 is 20.

```
        LDY  #20
.data   LDA ($70),Y
          .
        process
          .
          .
        DEY
        BPL  data
          .
        rest of program
```

The example should require little explanation, except perhaps to note that the indexing proceeds downwards towards zero, so the processing begins with the last data item and finishes with the first. As mentioned earlier, a downwards scan enables the end of the loop to be tested without the use of a CPY.

Some variations in the jargon exist. The alternative name for indirect indexed (and in some ways more informative) is 'post-indexed' indirect

addressing because the indexing is done after the indirect address has been found. Also, address pointers are sometimes called *address vectors*.

Indirect indexed addressing is available with ADC, AND, CMP, EOR, LDA, ORA, SBC, and STA. They all take 5 clock cycles except STA which takes 6. If a page boundary is crossed, they take an extra clock cycle.

Indexed indirect addressing

This mode doesn't enjoy quite the same measure of popularity as indirect indexed. The assembler form is:

```
LDA (operand, X)
```

Note carefully the position of the parentheses, that X is inside instead of outside and only X is allowed for indexing. As before, the operand must be single-byte so can only refer to a zero-page address.

X is shown within parentheses to emphasise the manner in which indexing is carried out. The behaviour of indexed indirect addressing is as follows:

> The address of the pointer in indexed indirect addressing is the sum of the operand and the contents of X.

This definition may explain why an alternative name of this mode is 'pre-indexed' indirect addressing. To aid understanding, first study the following numerical example:

```
LDA ($70,X)
```

In the first instance, assume that X is zero. The pointer is then the double byte address which happens to be in $70 (low-byte) and $71 (high-byte). However, if we assume that X contains $02, the address pointer is located at the double-byte address $72 and $73. Proceeding with this example, suppose that $72 contains $35 and $73 contains $0D, the instruction would load the accumulator with the contents of address $0D35. The example is illustrated in Fig. 3.3.

Until familiarity is gained, it is easy to get mixed up with the two indirect modes because of the relatively superficial differences in the assembler form. In order to emphasise the difference in form and effect, it is worth viewing the two side by side:

> *Indirect indexed* (post-indexed indirect) addressing keeps the pointer at a constant location but uses Y indexing to modify the pointer value.

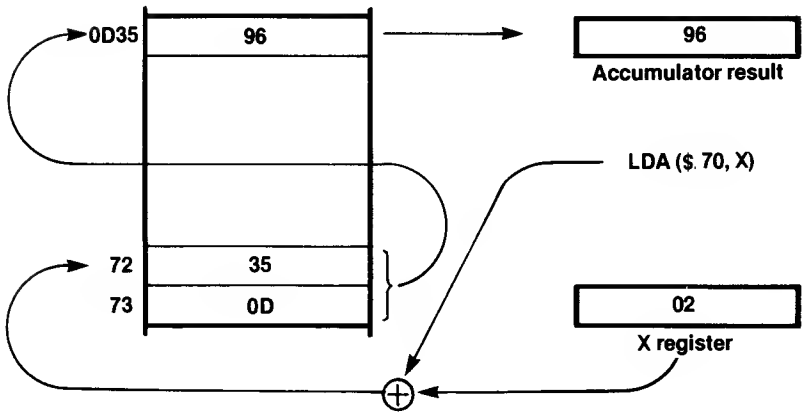


Fig. 3.3. Data flow in indexed indirect addressing.

Indexed indirect (pre-indexed indirect) addressing uses X indexing to modify the operand, and hence, the location of the address pointer.

As hinted earlier, indexed indirect addressing is not a commonly used mode. One area in which it is valuable is in handling peripheral interrupts. The course of a program can often depend on the particular peripheral which has requested interrupt. For example, the data sent to a printer will originate from a different area than the data sent to a digital-to-analogue converter. Assuming there are two peripherals on line, then we can arrange to have two separate address pointers to service them, located in zero-page. Suppose these double-byte addresses occupy the four locations $72, $73 and $74, $75 and consider the following line:

```
STA ($70,X)
```

The value placed in X must be that which modifies the operand to locate the desired address pointer. Care should be taken when calculating the value of X. The indirect address pointer is a two-byte address, so X must be changed by two at a time, otherwise the instruction above will define the high-byte instead of the low-byte. For example, if X is initially zero, the address pointer selected is located at $72, $73. If X is incremented only once, there is a foul-up because the address pointer is taken to be $73, $74 which is the high-byte of the first pointer and the low-byte of the second.

Apart from handling peripherals, indexed indirect addressing can be used to simulate the CASE statement found in some of the structured languages or the ON GOTO in BASIC. Control can be switched to separate machine code processes, each switched by a unique address pointer. The value in X determines which process is activated.

Indexed indirect addressing is available with ADC, AND, CMP, EOR, LDA, ORA, SBC and STA.

Summary

1. A machine code instruction always has an op-code but not all have operands.
2. The op-code defines the required action; the operand indicates where data is to be found.
3. Addressing modes are various ways in which operands express location of data.
4. The computer recognises only binary op-codes expressed as two hex digits but the resident assembler allows three-letter mnemonic groups.
5. The precise effect of an instruction is more concise if written in operational symbols rather than words.
6. During transfers, source data remains intact but old data at the destination is overwritten.
7. In normal use, the carry is cleared before adding but set before subtracting.
8. In double or multiple byte arithmetic, clear carry only before adding the lowest order bytes and set carry only before subtracting the lowest order bytes.
9. Memory or registers are cleared by a load zero. There are no CLR instructions.
10. There are no instructions to increment or decrement A.
11. Use AND to clear, ORA to set and EOR to change selected bits within a byte.
12. To flip over all bits, exclusive-or with $FF.
13. To produce two's complement, flip first and then add 1.
14. To find the state of a single bit, mask out uninteresting bits using AND and test for zero.
15. The BIT test copies bit 6 and 7 of the data into V and N bits respectively and ANDs the data into A.
16. LSR has the carry bit at the lsb end; ASR has the carry bit at the msb end.
17. Only shift and rotate instructions have accumulator-addressing.
18. In double-byte multiplication, use ASL for low-order and ROL for high-order byte.
19. In double-byte division, use LSR first for the high-order then ROR for the low-order byte.
20. The current state of the process status register determines whether or not a branch takes place.
21. Branch instructions themselves do not affect the process status register.
22. BMI and BPL are only useful if two's complement binary is used.

23. If the branch is out of range, combine with JMP.
24. In comparisons (CMP, CPX or CPY), the data is subtracted from the register in order to set flags but the original contents are restored.
25. To check if the register is less, use BCC; to check if equal use BEQ; to check if greater, use BEQ first then BCS.
26. Implied addressing has no operand.
27. Immediate addressing is when the operand, which must be single byte, is the data.
28. Absolute addressing is when the operand, which must be double-byte, is the address of the data.
29. Zero-page addressing is when the operand, which must be single byte, is the page-zero address of the data.
30. There are only 5 addresses guaranteed left free by the operating system, $FB to $FF inclusive.
31. Relative addressing, used only with branch instructions, is when the operand signifies how many bytes away is the next instruction.
32. Two's complement arithmetic is used to cover forward and backward branches. With the assembler, branch-to-label is possible.
33. Absolute indexed addressing is when the operand (which must be double byte) plus the index register, is the address of the data.
34. Zero-page indexed addressing is when the operand (which must be single byte) plus the index register, is the address of the data.
35. In an indexed instruction, the operand defines the base address, the index register the relative address. The sum of the two is the absolute or 'effective' address.
36. The operand in simple indirect addressing is the address of the lower order byte of a two-byte address pointer. Only JMP offers this mode.
37. JMP excepted, address pointers can only reside in zero-page (page-zero).
38. Indirectly indexed addressing modifies the address pointer by the addition of Y. The assembler operand format is (operand), Y.
39. Indexed indirect addressing modifies the address of the address pointer by the addition of the X register. The operand assembler format is (operand,X).
40. Address pointers are also called address vectors.

Self test

3.1 Should you clear or set the carry bit before subtracting?

3.2 If the accumulator holds $DF, what will it contain after the instruction EOR $DF has been executed?

3.3 Which of the three logical instructions are used to change selected bits within a byte?

3.4 What instruction could you use to ensure that bit 2 in the accumulator was in the 1 state, without disturbing the remaining bits?

3.5 One of these instructions is illegal, LDA #20, AND 20, STA #20. Which one?

3.6 Which type of instruction uses relative addressing?

3.7 What address is referenced if X contains $0F and we write STA $2003,X?

3.8 Only one instruction in the 6502 repertoire supports simple indirect addressing. Which one?

3.9 This is illegal: LDA ($30),X. Why?

3.10 What is wrong with this instruction: LDA ($0D45),Y?

3.11 If an assembler was not available, how could you write the instruction STA 35,X?

Chapter Four
Entering and Assembling Machine Code

Direct entry by POKE statements

Machine code bytes can be pushed directly into consecutive memory locations by means of a series of POKEs. In fact, if no additional software is purchased, this may be the only method possible. Once the program is successfully entered, the POKE method is as effective as any other but it must be admitted that the process is both tedious and error-prone. In fact, it would be difficult to imagine a more onerous task on a computer (any computer) than poking machine code into memory. Providing the routine is short, say less than 50 bytes, it might just be tolerable. On the other hand, the task of entering hundreds of bytes could turn out to be a traumatic experience, unlikely to be repeated in future unless the 'poker' is blessed with a cast-iron constitution.

The trouble with poking is the awful appearance of final listings. Row after row of numbers are spewed out, all quite meaningless unless each op-code byte is compared with the literal translation given in the 6502 instruction set. The POKE method is suitable for those who are content to enter programs listings from magazines or books. All that is needed for this operation is patience and care. But, when developing your own programs, there are many other pitfalls apart from entering code. The odds against a machine code program working first time are fairly high so debugging is an inevitable consequence after the initial entry. It is frustrating to develop and debug machine code without some form of software aid. However, in spite of these remarks, there will be many users who will be quite willing to put up with the difficulties and enter their code by a series of POKEs. It is simply a case of 'assembling by hand' and will involve the following steps:

(1) Write out the program on paper using the standard three-letter mnemonics for the operation codes together with the assembly format as given in Appendix C. In other words, proceed as if you were using the full assembler.
(2) Using the assembly code as a pattern, rewrite it with the correct hex codes and operand.
(3) Convert the hex into equivalent decimal (necessary because there is no provision in the computer for allowing direct entry in hex).

The final result should be a list of small decimal numbers none of which must exceed 255 decimal because this is the limit which can be entered with a single poke. For example, assume one of the assembly lines is as follows:

```
LDA #$21
```

We first note from the operand that it is using *immediate addressing* (because of the '#' character). We must then look up the correct LDA op-code for immediate addressing to find it is $A9. Since the operand is also given in hex, the original assembly code becomes

```
A9 21
```

The final stage is to convert this to decimal,

```
169 33
```

Using M for an arbitrary memory address, the POKE statement becomes

```
POKE M,169:POKEM+1,33
```

This example was deliberately chosen because it was simple. Not all assembly translation is quite so easy so it is worth trying out the following example which is a bit more tricky to follow:

```
LDA $FF34
STA $2510,X
```

Note that, in the first line, there are four hex digits in the operand. This indicates absolute addressing, so the LDA op-code is AD. Be careful with the operand because the 6502 expects the lower order byte of an absolute address to come first. Bearing this in mind, the complete instruction, written out in hex becomes

```
AD 34 FF (173 52 255 in decimal)
```

In the second line, the STA is using absolute index addressing with X as the index register, so the hex op-code is 9D and the operand with reversed bytes is 10 25. Therefore, the full line becomes

```
9D 10 25 (157 16 37 in decimal)
```

The complete POKE statements for the two lines becomes

```
POKE M,173:POKE M+1,52:POKE M+2,255:
POKE M+3,157:POKE M+4,16:POKE M+5,37
```

The above examples illustrate the emphasis the previous gloomy remarks regarding the entry of machine code by poking. The possibilities of error are too horrific to contemplate, particularly when lengthy programs are involved.

Where to load programs

It is customary to locate machine code in the special memory block reserved by the Commodore 64 operating system.

> Machine code programs can be safely stored in the 4K block commencing at address $C000 (49153 decimal).

Programs stored in this area are immune from the ravages of BASIC. They remain tucked away until called.

Programs to load machine code

It is possible to simplify the poking procedure by using a 'loading' program which can be written in BASIC. Program 4.1 is reasonably straightforward and worth saving on tape for use when entering hexadecimal machine code bytes. It can easily be adapted or expanded to suit individual taste.

```
10 REM POKING A HEX DUMP INTO MEMORY
20 REM STARTING AT ADDRESS $C000
30 INPUT"HOW MANY BYTES IN HEX DUMP";N%
40 B=49152
50 FOR L=0 TO N%-1
60 READ D$
70 FD%=ASC(D$)-48
80 SD%=ASC(RIGHT$(D$,1))-48
90 IF FD%>9 THEN FD%=FD%-7
100 IF SD%>9 THEN SD%=SD%-7
110 BT%=16*FD%+SD%
120 POKE B+L,BT%
130 NEXT
140 DATA A9,00,85,FB,A9,05,85,FC
150 DATA A9,48,20,CA,F1,38,A5,FB
160 DATA E9,01,85,FB,B0,02,C6,FC
170 DATA A5,FB,D0,EC,A5,FC,D0,E8
180 DATA 60
```

Program 4.1. Poking a hex dump into memory.

This program, controlled by a FOR/NEXT loop, allows the entry of machine code bytes directly in hex. This is helpful because it removes the error-prone activity associated with hex to decimal conversions.

Lines 10 to 130 form the actual program for performing the load. Lines 140 to 180 give sample DATA for purposes of illustration only. The program automatically places the code in the address $C000 onwards.

Line 30 asks for the number of bytes to be entered and is easy to answer if

you get into the habit of spacing your data bytes in blocks of eight per line. We shall see later that machine code monitors also stick to the idea of 8-byte blocks – 10 is an unfriendly number in machine code work.

Lines 70 to 110 take advantage of the fixed ASCII code interval between alphanumerics and zero to obtain a simple hex to decimal conversion. Lines 90 to 100 take care of the extra interval (7) between the ASCII code for '9' (57) and the ASCII code for 'A' (65) exclusive. By such means, the hex digits in the DATA blocks are neatly converted into decimal as demanded by the POKE statements at line 120.

It is convenient now to introduce the hex dump, a term used to describe a collection of hex digits displayed (or printed) with location addresses which represents the object code of a program. If you care to try out the program with the sample data supplied, you will find it displays on the screen the character 'H' 1024 times, with scrolling, of course, on the last 25 characters. Needless to say, having loaded the hex dump, it will need to be executed by using SYS 49152 (which is the decimal equivalent of $C000).

Entry by machine code monitor

A machine code monitor is a software aid to the task of entering machine code. The original Commodore PET series came with a very useful monitor called TIM built into the operating system but not so the 64. Fortunately, there are utility software packages available which include a TIM-like monitor. They will differ slightly in detail but, in general, such monitors will offer the facilities described below.

Display of memory block

Memory contents can be displayed, starting from a given absolute address in hex. The display is usually laid out in the form of blocks of eight memory locations together with the address of the first byte of each block. The contents of each location is shown in hex digit form. A typical display might have the following appearance:

```
$C000 DF 3C A9 24 35 FD 3F A4
$C008 34 57 AC B5 BD 47 20 00
$C010 20 00 3F BD 11 60 00 00
```

From this, we could deduce that address $C002 contains A9 and address $C00F contains 00. It is worth noting that actual programs will not necessarily use an exact multiple of eight bytes. This means that a few codes on the last line may be garbage. The last active code will probably be $60, the 6502 RTS instruction. The example above has two garbage bytes (00 00) following RTS.

Entering code

Having asked for the display of a block, as shown above, code is entered by moving the cursor over to existing digits and overwriting them with new code. This action automatically places the new contents into memory. Even if this were the only facility offered by a monitor, its use would be justified. The ability to use hex direct is a great help apart from the advantages of the block-of-eight display. Decimal is often an alien notation in machine code environments.

Register display

It is useful, and quite often necessary, to have a facility for viewing the current contents of the 6502 registers. A monitor will normally provide such a display at any time by keying 'R'. The registers displayed will be the Program Counter, Status Register, Accumulator and the X and Y registers. Overwriting existing register contents by means of the cursor and edit keys is often provided.

Assemblers

The ultimate software aid for machine code programmers is an assembler. In fact, it is doubtful whether any serious development work with machine code can be carried out without one. An assembler may be looked upon as a kind of super-monitor equipped with powerful extras. The most outstanding of these extras is an ability to 'understand' op-codes entered in the form of mnemonic letter groups instead of hex. The irksome task of looking up the instruction set for hex codes (which have no intelligible meaning to humans) is replaced by an instantly recognisable group of letters. Thus, a line like A9 #4 can be written as LDA #4.

If you look up Appendix C, you will notice that there are eight different LDAs, each with a separate pair of hex op-codes. An assembler sorts out which particular LDA you want by noting the format of your operand. In the example just quoted above, the particular LDA (the one which uses immediate addressing) is chosen by the assembler because the operand is only two decimal digits and is preceded by the character '#'. It is evident that, even with an assembler, the exact format required by the assembler for each addressing mode must be respected.

Assembly languages

We have, of course, been using assembly language freely during the lead up to this but it is now time to enquire if all assembly language is the same. Fortunately, it is possible to give a tentative 'yes' to this question providing we are talking about the same microprocessor. That is to say, assembly language for the 65XX microprocessor family (which includes, of course, the 6502 and the 6510A) is, from the user's point of view, virtually the same

in all machines which employ them. This is because the mnemonic letter groups and the operand format for different addressing modes are defined rigidly by the manufacturers when new families of microprocessors are first launched. Although this leads to a degree of standardisation, it is unreasonable to expect that an assembler written for machine X will work on machine Y just because they both employ the same microprocessor. Apart from the microprocessor, there may be differences in the operating system and almost certainly in RAM capacity and address allocations, both of which rule out the chance of compatibility with other machines.

The MIKRO 64 Assembler

Whenever a microcomputer appears to be gaining popularity, one or more of the established software houses will bring out an assembler for it. This has happened in the case of the Commodore 64. Although Commodore markets a wide range of in-house software aids, including an assembler, other firms have also launched their own versions. The assembler we have used during the production of this book has been the MIKRO 64 Assembler. We have found it to be a powerful assembler which is easy to use. However, it would be wrong to imply from this that the MIKRO 64 is in any way superior to the official Commodore version, or indeed, any other assembler on the market. Nevertheless, the operating instructions which follow are valid for the MIKRO 64 and based on the handbook accompanying the assembler cartridge. The instructions for other assemblers may differ perhaps in slight detail and facilities offered but not in overall structure.

Assembler passes

MIKRO 64 is defined as a 'three pass assembler', meaning that three scans of the source code, written in assembly language are required before the end result (pure machine code) resides in memory. More than one pass is required because symbolic operands and jump labels must be assigned to absolute addresses. The assembler performs the three passes automatically without assistance from the user.

MIKRO 64 Assembler notation

To list all the formatting rules would be to repeat what is already in the accompanying manual so it is sufficient here to show a simple program to illustrate some of the more important features.

This is how a program may appear as initially typed:

```
10 *=$C000
20 SCREEN=$0400
30 LDX #0
40 LDA #0
```

```
50 BACK STA SCREEN,X
60 INX
70 BNE BACK
80 RTS !BACK TO BASIC
```

Line 10 illustrates how to tell the assembler where to locate the first byte of the assembled code. Note that * followed by = is mandatory at the left. The ultimate effect of this line will be to set the Program Counter to start at $C000.

Line 20 shows how to allocate a symbolic name to an absolute memory address. Once set like this, all subsequent references to SCREEN will be taken by the assembler to mean $0400 which, incidentally, is the address of the top left-hand corner of the screen.

Lines 30 and 40 are obvious.

Line 50 should be studied carefully because it illustrates the format for branch labels. The first word 'BACK' is a branch label. How does the assembler know this? Because it first searches through its dictionary to see if the word is a legal mnemonic op-code. If not, it searches again to see if it is a legal pseudo-op. Only after these two searches have been proved false does it then conclude that the word must be a branch label. The operand, in line 50, indicates that simple indexed addressing is required.

Line 70 is the bottom of the loop containing the branch to the label BACK.

The last line, containing RTS, shows the method of appending remarks by preceding them with '!'.

Spaces and FORMAT command

At least one space to separate the different parts of the code on a line is of fundamental importance. They should follow the pattern above. If you leave them out the assembly process will fail. More than one space can be left between the parts but there is no need to use them to make the assembly listing more presentable. To present a nice clear listing on paper, the assembler is equipped with a FORMAT command. To see how this works, we assume that the example program given earlier has been assembled and we now require a posh printout. The procedure for printers connected to a parallel interface is:

```
OPEN6,6:CMD6:FORMAT
```

The result should appear as follows:

```
10 *=$C000
20 SCREEN     =   $0400
30            LDX #0
40            LDA #0
50 BACK       STA SCREEN,X
60            INX
70            BNE BACK
80            RTS !BACK TO BASIC
```

Listings in this book have been printed under the FORMAT command because of the resulting clarity. The wide separation between labels and mnemonic op-codes aids comprehension. There is only one snag. The listings must be in camera-ready form for publishing which means that, for technical reasons, the maximum width should preferably be limited to 100 mm. This does not leave much room for remarks so they have been left out altogether from our programs. In any case, remarks on listings are necessarily staccato in style and can often pollute, rather than clarify. Consequently, programs in this book are explained in the accompanying text.

The OUT command

The OUT command is used to output data to the screen or printer. OUT,4 will direct output to the printer connected to the RS232 serial interface and OUT,6 to the parallel interface. If the command is given without a device number, the output is directed to the screen by default. The data outputted is that which follows the OUT command and continues until the command OFF or END appears.

Operand format

This is standard 6502 format but, to aid continuity, some examples, using LDA as the mnemonic op-code, are given below:

LDA #32	immediate (decimal)
LDA #$32	immediate (hex)
LDA $FF	zero-page
LDA $40BF	absolute
LDA BLOGS	absolute
LDA $D0,X	zero-page indexed
LDA MEMORY,Y	absolute indexed
LDA (POINTER),Y	indirect indexed
LDA (POINTER,X)	indexed indirect

Pseudo-ops

Pseudo-ops are instructions, given in source code, to the assembler. This distinguishes them from true op-codes recognisable by the microprocessor. All assemblers have a few of them. The MIKRO 64 assembler includes the following pseudo-ops in its repertoire:

WOR...can be used to place 16-bit data values into memory in low-high byte form. For example,

```
10 *=$C000
20 WOR 65535,$4867,$0653
```

This would store in successive locations, starting from address $C000, the bytes $FF,$FF,67,48,53 and 06.

BYT...is similar to WOR but used to store 8-bit data. For example,

```
20 BYT $25,254,16
```

This would store the bytes $25,$FE,$10. To save working out ASCII values, BYT will also convert characters to their ASCII equivalent, providing the character is preceded by '. For example, BYT $23,'A,20 would store the bytes $23,$41,$14.

TXT ... can be used to store text characters in successive locations. For example:

```
50 TXT "GREASBY,MERSEYSIDE"
```

This will store the eighteen ASCII characters within the double quotes. Program 4.2 includes an example of the use of TXT. The first line of the program assigns the name OUTPUT to the kernal subroutine CHROUT located at address $FFD2 (see Appendix B). The individual characters of the text SHORT WAVE RADIO, 16 in all including the spaces, are sent one at a time to the OUTPUT subroutine by indexed addressing in line 40.

```
 10 OUTPUT        =   $FFD2
 20 *=$C000
 30               LDX #0
 40 LOOP          LDA TEXT,X
 50               JSR OUTPUT
 60               INX
 70               CPX #16
 80               BNE LOOP
 90               BEQ OVER
100 TEXT          TXT "SHORT WAVE RADIO"
110 OVER          RTS
```

Program 4.2. Use of pseudo-op TXT.

The next, Program 4.3, uses the pseudo-op BYT for laying down sound output data.

The program is fairly straightforward and hardly justifies detailed line-by-line treatment. The data table uses BYT for storing the pitch and envelope parameters. They have been selected to sound the eight individual notes which span an octave of the major diatonic scale in C major. The envelope parameters are intended to simulate the percussion tones of a piano. Delay between the notes is accomplished by calling the subroutine DELAY, occupying lines 380 to 430. The loop counter, determining the delay time, is initialised to $FB and assigned to the variable DUR at line 40.

```
 10 ! C MAJOR SCALE WITH
 20 ! ADSR ENVELOPE SHAPING
 30 BASE          =  $D400
 40 DUR           =  $FB
 50 *=$C000
 60               LDX #24
 70               LDA #0
 80 LOOP          STA BASE,X
 90               DEX
100               BPL LOOP
110               LDA #9
120               STA BASE+5
130               LDA #0
140               STA BASE+6
150               LDA #15
160               STA BASE+24
170               LDX #0
180 LOOP2         LDA TABLE,X
190               STA BASE+1
200               INX
210               LDA TABLE,X
220               STA BASE
230               INX
240               LDA TABLE,X
250               STA DUR
260               LDA #33
270               STA BASE+4
280               JSR DELAY
290               LDA #32
300               STA BASE+4
310               LDA #255
320               STA DUR
330               JSR DELAY
340               INX
350               CPX #24
360               BNE LOOP2
370               RTS
380 DELAY         LDY DUR
390 LOOP3         LDA $D012
400               BNE LOOP3
410               DEY
420               BNE LOOP3
430               RTS
440 TABLE         BYT 16,195,100,18,209,100
450               BYT 21,31,100,22,96,100
460               BYT 25,30,100,28,49,100
470               BYT 31,165,100,33,135,255
```

Program 4.3. C major scale.

Renumbering

Unfortunately, MIKRO 64 does not provide a line renumbering program, probably because the designers felt it was unnecessary. Line numbers in

```
 10 ! FIXED RENUMBER ROUTINE
 20 ! FOR MIKRO 64 ASSEMBLER
 30 ! START LINE 10 INCREMENT 10
 40 LINK          =  $FB
 50 LINE          =  $FD
 60 *=$C400
 70               LDA #10
 80               STA LINE
 90               LDA #0
100               STA LINE+1
110               LDA $2B
120               STA LINK
130               LDA $2C
140               STA LINK+1
150 LOOP          LDY #0
160               LDA (LINK),Y
170               PHA
180               INY
190               LDA (LINK),Y
200               TAX
210               LDA LINE
220               INY
230               STA (LINK),Y
240               LDA LINE+1
250               INY
260               STA (LINK),Y
270               CLC
280               LDA LINE
290               ADC #10
300               STA LINE
310               BCC SKIP
320               INC LINE+1
330 SKIP          STX LINK+1
340               PLA
350               STA LINK
360               BNE LOOP
370               TXA
380               BNE LOOP
390               RTS
400               END
```

Program 4.4. Renumber routine.

BASIC are important because GOTO destinations refer to line numbers. In contrast, line numbers in assembly listings have no such status and are useful only for references purposes in textual descriptions. Nevertheless, there is no denying that a listing with ragged line numbers is not a pleasant sight. A listing with line numbers 10,11,13,47,49, etc. does not inspire confidence. There is always a feeling (probably an unfair one) that the program has been hastily put together. Program 4.4 is a neat renumbering routine which can be used before printing out a final assembly listing. It is a modified version of a program taken from Ian Sinclair's excellent book *Introducing Commodore 64 Machine Code* (Granada).

Machine code monitor

The MIKRO 64 assembler is also equipped with a machine code monitor and, from the user's viewpoint, is similar to the original Commodore TIM but including some powerful extras. To call up the monitor, it is sufficient to enter:

```
CALL @814B
```

The response on the screen is a simple register dump as follows:

```
ADDR IRQ SR XR YR SP
.;814B EA31 3B 00 00 F9
```

The abbreviations refer to the 6510A internal registers, the interrupt vector (IRQ), the status register (SR), the X and Y registers (XR,YR) and the stack pointer (SP).

The syntax for calling up monitor facilities is quite straightforward:
Display registers: enter R.
Display memory block: enter M followed by the hexadecimal start and end addresses. (*Example:* R C000 C0FF)
Save on disk: enter S "drive:name",device number,start address,end address +1. (*Example:* S "0:TEST"08,C000,C100)
Load from disk: enter L "name",08. (*Example:* L "TEST",08)
Note: If the device number is not quoted, saving or loading takes place on cassette.
Execute program: enter G alone or G followed by the start address. G alone will execute from the program counter address shown in the register display. (*Example:* G C004)
Hunt for byte pattern: enter H, followed by the start and finishing addresses of the block to be searched, followed by the byte pattern. (*Example:* C000 C0FF 30 C3 7D)
Transfer memory block: enter T followed by start and finishing address+1, followed by the start address of the new block. (*Example:* T C000 C100 D000)

Disassemble: enter D followed by start and finishing addresses of the block to be dissassembled. (*Example:* D C000 C0FF)
Note: Illegal bytes are shown as BYT.

Free zero-page locations

The Commodore 64 is not over-endowed with user-free zero-page locations. In fact the only ones guaranteed to be vacant are the 5 bytes from $FB to $FF. This can be a handicap with larger programs since indirect indexed addressing must use zero-page locations. Fortunately, there is a way out, because we can reclaim most of the zero-page BASIC workspace as long as the locations are used only within the machine code routine itself. This is quite in order as long as it is remembered that the locations may be overwritten at any time on returning to BASIC. Most of the programs in Chapters 6 and 7 employ this technique without any adverse problems. However, it is best where possible to use $FB to $FF for passing parameters between BASIC and machine code routines and vice versa.

Zero-page locations free:

$FB to $FF

Zero-page BASIC workspace that can be reclaimed:

$4E to $60
$26 to $29

There are a few other locations here and there which can be reclaimed, but the above is more than sufficient for all the programs in this book.

Where to obtain the MIKRO 64 Assembler cartridge
The MIKRO 64 Assembler is distributed by:

Supersoft
Winchester House
Canning Road
Wealdstone
Harrow HA3 7SJ
England

In North America the MIKRO 64 Assembler is distributed by:

Skyles Electric Works
231E S Whisman Road
Mountain View
CA 94041
USA

Summary

1. Machine code can be entered by poking instruction bytes directly into memory.
2. Direct poking requires looking up hexadecimal op-codes and then converting to decimal.
3. Directly poked 2-byte operands must be entered in low-byte, high-byte order.
4. Directly poked decimal numbers must not exceed 255.
5. The safe area for locating machine code programs is the 4K block from $C000 onwards (49153 decimal).
6. Program 4.1 is a useful hex loading program, allowing hex bytes to be used.
7. A machine code monitor, if subsequently installed, simplifies entering and debugging machine code.
8. An assembler is the supreme software aid to all machine code work. Without one, serious development work is hardly possible.
9. The programs in this book have been developed with the aid of the MIKRO 64 Assembler.
10. At least one space must be left between fields. More than one is allowed.
11. The appearance of the listings in this book are due to the FORMAT command.

Self test

4.1 What is the largest decimal number you can POKE?

4.2 What POKE numbers would you use to enter the machine instruction AND #$32?

4.3 What POKE numbers would you use to enter the machine instruction LDA 256?

4.4 The last POKE number in a machine code block is often 60. Why?

4.5 State an important facility present in an assembler but not in a monitor.

4.6 Using the MIKRO assembler, write the line which positions the starting address at $C234.

4.7 How does a pseudo-op differ from a normal machine op-code?

4.8 Is WOR a genuine op-code or a pseudo-op?

4.9 In the MIKRO assembler, what is the pseudo-op for storing text characters?

Chapter Five
Machine Code Building Bricks

Bricks versus instructions

During the initial learning phase, it is a bewildering task trying to work out the coding required to perform even the most simple operation. As experience is gained, you notice that certain code *patterns* seem to crop up over and over again. In other words, you begin to recognise the existence of general purpose building bricks which will conveniently fit into program slots. You no longer think in terms of individual instructions because the building bricks, rather than the individual (and primitive) machine code instructions, become the 'atoms' of action. It becomes a question of which building brick to choose, rather than which instruction to choose.

When you reach this desirable situation machine code changes from the horrific to the benign. It is rather like a child learning to read. During the first week or two, reading is a painfully hard and laborious process because only the individual letters are recognised and each word has to be built up from them. Eventually, groups of letters (words) are recognised on sight and the mind is barely conscious of the individual letters which form them. Speed readers advance even further and can comprehend entire lines at sight without even being consciously aware of the words.

However, there is one danger if the analogy between learning machine code and reading is taken too literally. Machine code building bricks are ideal as preliminary inserts whilst the program is passing through the development stage. But, because of their general-purpose nature, they would normally require polishing up before the program is finally released. Nothing designed to be general-purpose in its functions can at the same time be of maximum efficiency when specifically employed. The polishing up stage would normally entail removing code which has been found to be redundant because of *pre-existing* code. For example, the building brick in its original general-purpose form may have included CLC at the top but, when employed in the program slot, the carry might already have been cleared. Of course, the opposite condition can also occur. It may be that the building brick requires an extra line or two in order to tailor it to fit a specific situation.

In spite of these occasional variations, there is much to be said for acquiring the building brick habit as soon as possible. The rest of the chapter is devoted to this subject and although it includes many examples, you are strongly advised to add to the list. In this way, a fairly exhaustive 'library' of them can be formed, able to cope eventually with the most bizarre requirements.

Machine code equivalents to BASIC

Most of us begin computer studies within the comfort of BASIC rather than beginning with, and risking the perils of, machine code. In view of this, it helps to sweeten the pill if a gentle transition to machine code is made via roughly equivalent BASIC forms. The following section describes some common BASIC structures with their machine code equivalents. As an initial simplification, the machine code equivalents are restricted to single byte, two's complement working.

Assigning constants:

BASIC	Machine code
SPEED=30	LDA #30 STA SPEED

Re-assigning variables:

BASIC	Machine code
S=B	LDA B STA S

Adding a constant:

BASIC	Machine code
A=A+23	LDA A CLC ADC #23 STA A

Subtracting a constant:

BASIC	Machine code
A=A−15	SEC LDA A SBC #15 STA A

Addition and subtraction:

BASIC	Machine code
A=A+K−8	LDA A CLC ADC K SEC SBC #8 STA A

Doubling a number:

BASIC	Machine code
N=N*2	ASL N

Expressions:

BASIC	Machine code
N=4*(K+25)	LDA K
	CLC
	ADC #25
	ASL A
	ASL A
	STA N

Incrementing by 1:

BASIC	Machine code
N=N+1	INC N

Decrementing by 3:

BASIC	Machine code
N=N−3	DEC N
	DEC N
	DEC N

Calling subroutine:

BASIC	Machine code
GOSUB 2460	JSR BLOGS

Note that whereas in BASIC the GOSUB is to a meaningless line number, a machine code assembler allows a meaningful label to be used instead of working out the destination line number.

Returning from subroutine:

BASIC	Machine code
RETURN	RTS

Simple loop:
Writing the equivalent machine code versions of FOR/NEXT loops is fairly straightforward. The only hazard is remembering that one extra loop revolution is demanded before exiting. This extra 'one' can either be allowed for at the beginning or, as in the following examples, at the end where the final comparison is made.

BASIC	Machine code
FOR N=1 TO 20	LDX #1
	BACK xxxx
	.
NEXT	.
	INX
	CPX #21
	BNE BACK

Simple loop with step:

BASIC	Machine code
FOR N=0 TO 80 STEP 5	LDX #0
.	BACK xxxx
.	.
NEXT	.
	CLC
	TXA
	ADC #5
	TAX
	CPX #85
	BNE BACK

Loop with variables:

BASIC	*Machine code*
FOR N=S TO F−1	LDX S
.	BACK xxxx
.	.
NEXT	.
	INX
	CPX F
	BNE BACK

Loop with step:

BASIC	Machine code
FOR N=S TO F−1 STEP J	LDX S
.	BACK xxxx
.	.
NEXT	.
	CLC
	TXA
	ADC J
	TAX
	CPX F
	BNE BACK

Decrementing loop:

BASIC	Machine code
FOR N=10 TO 1	LDX #10
.	BACK xxxx
.	.
NEXT	.
	DEX
	BNE BACK

Note that a decrementing loop is easier and does not require either a CPX or an allowance for an extra 'one'.

Branch if zero:

BASIC	Machine code
IF N=0 THEN GOTO 500	LDA N
	BEQ LABEL
	.
	.
	LABEL xxxx

Branch if N non-zero:

BASIC	Machine code
IF N<>0 THEN GOTO 500	LDA N
	BNE LABEL
	.
	.
	LABEL xxxx

Branch if N<0:

BASIC	Machine code
IF N<0 THEN GOTO 500	LDA N
	BMI LABEL
	.
	.
	LABEL xxxx

Branch if >= zero:

BASIC	Machine code
IF N>=0 THEN GOTO 500	LDA N
	BPL LABEL
	.
	.
	LABEL xxxx

Note that zero is classified as a positive number.

Change contents of variable if zero:

BASIC	Machine code
IF N=0 THEN S=Q	LDA N
	BNE LABEL
	LDA Q
	STA S
	LABEL xxxx

If zero, change sign of a variable:

BASIC
```
IF N=0 THEN S=-S
```

Machine code
```
LDA N
BNE LABEL
LDA S
EOR #$FF
CLC
ADC #1
STA S
LABEL xxxx
```

Branch if both comparisons are true:

BASIC
```
IF N=30 AND K=22
THEN GOTO 500
```

Machine code
```
LDA N
CMP #30
BNE LABEL1
LDA K
CMP #22
BEQ LABEL2
LABEL1 xxxx
.
.
LABEL2 xxxx
```

Branch if either of two comparisons are true:

BASIC
```
IF N=30 OR K=22
THEN GOTO 500
```

Machine code
```
LDA N
CMP #30
BEQ LABEL
LDA K
CMP #22
BEQ LABEL
.
.
LABEL xxxx
```

Double byte working

Single byte working is ideal for illustrating the basic principles of the 6502 or, indeed any other 8-bit microprocessor. However, machine code programs of practical value must assume that numbers will, in many cases, greatly exceed the capacity of a single byte. We should always bear in mind that the maximum unsigned binary number in a byte is limited to 255

decimal ($FF). With signed numbers in two's complement form, the maximum positive number is +127 and −128 decimal ($7F and $80) respectively. These limits are dictated by the hardware but, fortunately, it is easy to overcome them by the application of software. Multi-byte (or multi-precision) working is the software solution. In other words, an 8-bit microprocessor can, by software, simulate a microprocessor of (theoretically) any desirable wordlength.

There are penalties, of course – the most important being increased execution time and extra programming involved in arranging the component bytes. The examples which follow in this chapter are intended only for guidance, not as fully workable programs. To have made them workable would have entailed adding various initialising lines, thus complicating, rather than clarifying, the issue.

Handling two-byte numbers

By considering one number as two bytes joined end to end, the maximum unsigned binary number increases to 65535 ($2^{16}-1$). In two's complement binary, the maximum positive number is 32765 ($2^{15}-1$) and the maximum negative number is −32766 (2^{15}). It does not matter in principle where the component bytes are stored during processing but it is logical for them to be stored in consecutive locations with the lower order byte first. For example, we might quite arbitrarily choose the address $1234 for the lower order byte so the highest order byte would naturally reside in $1235. However, it is customary in any explanatory text involving locations to avoid specifying absolute addresses. It is far better to choose a variable name, preferably of mnemonic value, even if it is only provisional. Thus we could refer to the composite two bytes as NUMBER but, with regard to the addresses, the lower order byte would be located in NUMBER and the higher order byte in NUMBER+1.

Incrementing a two-byte number

The numerical limit imposed by a single byte is also a nuisance in loop counting. For example, we can't set up a simple loop to clear the screen if the total number of revs is limited to 256. However, it is easy to extend this by using a couple of extra instructions in the counting loop as shown in the following example:

```
INC NUMBER
BNE SKIP
INC NUMBER+1
SKIP xxxx
```

NUMBER is the low order byte of the loop counter and NUMBER+1 the high order byte. While the count remains less than 255, only the low order byte is incremented because of the branch to the branch label SKIP (XXXX indicates any instruction which would be applicable in a real case).

On the 256th revolution, NUMBER goes over the top from 255 to zero. When this occurs, the branch to SKIP is not taken and the higher order byte of the counter in NUMBER+1 is incremented ready for the next round of the outer loop. Note that the inner loop will revolve 256 times for each revolution of the outer loop. The outer loop can, if desired, revolve 256 times, making the complete count equal to 256×256 or 65536 revs. However, the outer loop test (not shown) can be tailored to suit any required number of revolutions within that limit.

Decrementing a two-byte number

It is not quite so straightforward to set up a two-byte loop which decrements towards zero rather than incrementing towards some finite number. This is because SBC must be used instead of DEC to avoid trouble with the carry bit.

The following procedure is as economical (in execution time) as any:

```
SEC
LDA NUMBER
SBC #1
STA NUMBER
BCS SKIP
DEC NUMBER+1
SKIP xxxx
```

Note that SBC is used for decrementing the low order byte instead of DEC. This is because

(a) DEC will not effect the carry flag
(b) The Z flag cannot be used because the high-byte is only decremented when the low-byte has passed through zero.

Adding two single byte numbers

It is obvious that when adding two single byte numbers, although the numbers may be within the capacity of a single byte, we must allow for the possibility of a double byte result. In the following example, the double byte result is assumed to be in SUM and SUM+1.

```
LDA #0
STA SUM+1
CLC
LDA NUMBER1
ADC NUMBER2
STA SUM
BCC SKIP
INC SUM+1
SKIP xxxx
```

Adding a single byte number to a double byte number

```
CLC
LDA NUMBER1
ADC NUMBER2
STA NUMBER1
BCC SKIP
INC NUMBER1+1
SKIP xxxx
```

The double byte number is NUMBER1 and NUMBER1+1 and the single byte number is NUMBER2. The new result overwrites the original data in NUMBER1, a practice frequently used. For example, in BASIC, A=A+1 employs a similar overwrite technique. Extra locations for the result must be used if the original data is to be preserved.

Adding two double byte numbers

```
CLC
LDA NUMBER1
ADC NUMBER2
STA RESULT
LDA NUMBER1+1
ADC NUMBER2+1
STA RESULT+1
```

Subtracting a single byte number from a double byte number
The coding for subtraction is similar to addition except that SEC and SBC are substituted for CLC and ADC respectively.

```
SEC
LDA NUMBER1
SBC NUMBER2
STA NUMBER1
BCS SKIP
DEC NUMBER1+1
SKIP xxxx
```

Subtracting a double byte number from a double byte number

```
SEC
LDA NUMBER1
SBC NUMBER2
STA RESULT
```

```
LDA NUMBER1+1
SBC NUMBER2+1
STA RESULT+1
```

Multiplication by a constant

The 6502, in common with nearly all other 8-bit microprocessors, does not know how to multiply or divide. It has no MUL or DIV instruction in its repertoire. When working in BASIC, this deficiency is not noticed because the interpreter has supplied multiplication and division subroutines. However, because these subroutines are general-purpose, and must include multiplication and division of floating point numbers of enormous magnitude, they, quite naturally, tend to be a little on the slow side. In contrast, the majority of numbers encountered in everyday life (or in data processing) are of unassuming size.

Because of this, it is worth considering a few multiplication routines even if they are restricted to small numbers. Subject to certain restrictions, a simple way to multiply, or divide, is by the use of shift and rotate instructions.

Multiplying a two byte number by a constant power of 2

```
ASL LOWBYTE
ROL HIGHBYTE
```

Shifting a register (or memory location) left is equivalent to multiplying by 2. In the case of a double byte number, we must first use ASL to shift the lower order byte then follow it with ROL which, in addition to the natural shift action, collects any carry generated by the previous ASL.

In general, each time we repeat the above action, we double the previous value. For example, the following code multiplies by 4:

```
ASL LOWBYTE
ROL HIGHBYTE
ASL LOWBYTE
ROL HIGHBYTE
```

Multiplying a two byte number by non-integral powers of 2

It is not possible to multiply by 3,5,6,7,9, etc., because they are not exact integrals of powers of 2. Consequently, we must resort to a piecemeal procedure involving shifting and adding. The following example will

multiply the contents of NUMBER and NUMBER+1 by 3 and store the product in RESULT and RESULT+1.

```
LDA NUMBER+1
PHA
LDA NUMBER
ASL A
ROL NUMBER+1
CLC
ADC NUMBER
STA RESULT
PLA
ADC NUMBER+1
STA RESULT+1
```

To multiply a two byte number by 5
The following code is similar to, but more lengthy than, the above.

```
LDA NUMBER+1
PHA
LDA NUMBER
ASL A
ROL NUMBER+1
ASL A
ROL NUMBER+1
CLC
ADC NUMBER
STA RESULT
PLA
ADC NUMBER+1
STA RESULT+1
```

It would, of course, be possible to multiply by repeated addition and avoid the complexity of shifting and adding but it could be at the expense of coding. When the multiplying constant is large, say greater than 5, then loop methods would be employed.

Branching and testing

The building brick idea can be extended to cover branching and end-of-loop testing. These are the most error-prone techniques in machine code. The most common error is being one count out in the number of revolutions.

Comparing a two-byte number to zero and branching if result is non-zero

The two-byte number to be compared is assumed to be in NUMBER and NUMBER+1

```
LDA NUMBER
BNE LABEL
LDA NUMBER+1
BNE LABEL
```

Comparing a pair of two byte numbers and branching if they are unequal

```
LDA NUMBER1
CMP NUMBER2
BNE LABEL
LDA NUMBER1+1
CMP NUMBER2+1
BNE LABEL
```

Up counting, using a twin byte loop

All previous examples in this chapter have been presented in terse format, intended for study rather than for keyboard entry. Program 5.1 is a simple example of an up counting loop which will fill the screen with the character 'H' and a few over for luck. It is worth entering, if only to consolidate the idea behind building bricks.

```
 10 ! UPCOUNTING TWIN BYTE LOOP
 20 OUTPUT        =  $FFD2
 30 CYCLE         =  $FB
 40 NUMBER        =  $FD
 50 *=$C000
 60               LDA #0
 70               STA NUMBER
 80               STA CYCLE
 90               STA CYCLE+1
100               LDA #4
110               STA NUMBER+1
120 LOOP          LDA #$48
130               JSR OUTPUT
140               INC CYCLE
150               BNE SKIP
160               INC CYCLE+1
170 SKIP          LDA NUMBER
```

```
180                CMP CYCLE
190                BNE LOOP
200                LDA NUMBER+1
210                CMP CYCLE+1
220                BNE LOOP
230                RTS
```

Program 5.1. Up counting twin byte loop.

The first keyable program in a chapter deserves a line-by-line treatment:

Line 10: the character '!' indicates that what follows is *comment* only.
Line 20 assigns the arbitrary chosen label OUTPUT to the absolute address $FFD2. This is the address of the kernal subroutine CHROUT which prints the character corresponding to the ASCII code in the accumulator.
Line 30: the label CYCLE is assigned to the zero-page address $FB. This is the low-byte address of the counter; the high-byte will be addressed by CYCLE+1 (which of course is $FC).
Line 40: the number of characters to be printed is in NUMBER (low-byte is assigned to $FD so high-byte is in NUMBER+1 at $FE).
Line 50: the first byte of the assembled program is to start at $C000. This format is standard to the MIKRO Assembler.
Lines 60 to 90 initialise the low byte of NUMBER, CYCLE and CYCLE+1 to zero.
Lines 100 and 110 load 4 into the high byte of NUMBER. (Note that the two byte number is now 400 in hex or $4 \times 256 = 1024$ in decimal.) This will be the final end of loop comparison test prior to exit.
Line 120 loads the accumulator with the ASCII code for 'H'. The line is also the top of the loop so has been given the branch label LOOP.
Line 130 calls the subroutine for printing the character on the screen.
Lines 140 to 160 increment CYCLE and, where necessary, CYCLE+1. Note that only when CYCLE has passed the 255 barrier and gone over the top to zero is CYCLE+1 incremented. Until this state arrives, the program will always short-circuit to SKIP.
Lines 170 to 220 form the end of loop test by comparing the low bytes of CYCLE and NUMBER and branching back to LOOP if not equal. If they are equal the program falls through to compare the high bytes, again branching back to LOOP if not equal.

Figure 5.1 shows a flowchart of the program but the 'process' is kept undefined to maintain the building brick concept. The 'process' in the listing (painting 'H's over the screen) was intentionally trivial but it should be clear that lines 120 and 130 can be replaced and expanded to cover any repetitive process. The point of the program was to portray the loop mechanism, not the process within the loop.

Although loops which *count up* towards a terminating number always

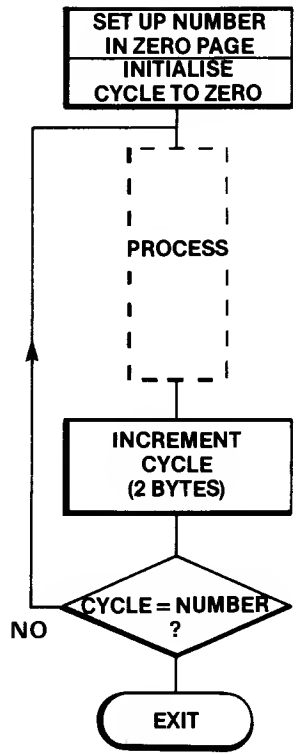


Fig. 5.1. Flowchart for Program 5.1.

seem more natural, they require an extra comparison test and so are slightly less efficient than down counting loops.

Down counting using a twin byte loop

Program 5.2 can also be keyed in direct from the listing shown. Since this

```
 10 !DOWNCOUNTING TWIN BYTE LOOP
 20 OUTPUT        =   $FFD2
 30 CYCLE         =   $FB
 40 *=$C000
 50               LDA #0
 60               STA CYCLE
 70               LDA #4
 80               STA CYCLE+1
 90 START         LDA #$48
100               JSR OUTPUT
```

```
110              SEC
120              LDA CYCLE
130              SBC #1
140              STA CYCLE
150              BCS SKIP
160              DEC CYCLE+1
170 SKIP         LDA CYCLE
180              BNE START
190              LDA CYCLE+1
200              BNE START
210              RTS
```

Program 5.2. Down counting twin byte loop.

program is broadly similar to the previous program, a line-by-line treatment was considered unnecessary. However, the change in counting direction allows two economies. In the first case, there is no need for the variable NUMBER and NUMBER+1. Instead, CYCLE is decremented directly and the loop termination is compared to zero.

User subroutines

Machine code programs, called from, and intended to return to, BASIC via RTS are essentially subroutines. However, it is a common requirement for the machine code program itself to use subroutines, either user-designed or one of the many resident subroutines embedded within the kernal operating system. Subroutines designed by the user are called by JSR followed by an operand, either an absolute machine address (not recommended) or a destination *label.*

As in BASIC, machine code subroutines can be nested one within the other. Enthusiasm for high nesting levels should not be carried to excess or the stack could overflow. Each unreturned JSR uses up two stack locations, storing the two-byte return address in the Program Counter. No provision is made in the 6502 for saving the other registers. It is up to the programmer to make provisions for protecting valuable register data from corruption by the subroutine. Subroutines are best avoided altogether within loops which are time critical. Each JSR squanders 6 clock cycles and RTS another six. It is far better to splice the code within the main program, even if it means writing the same segment of code several times.

Subroutines in the kernal

There are many subroutines buried within the kernal, most of which can be

called up by the user. They constitute a valuable source of 'free' building bricks. It would be pointless at this stage to plod through the entire repertoire of kernal operating system calls. The complete list appears in the Commodore 64 Programmer's Reference Guide but, for convenience, has been repeated in slightly different form in Appendix B at the back of this book.

Jump vectors

As far as the user is concerned, the kernal is essentially a jump table containing a set of calling addresses for the various subroutines. Most are called via a direct jump, but some of the more useful subroutines are called via an indirect jump. We have already used one of these, CHROUT, which had the calling address $FFD2 or 65490 decimal. This uses an indirect jump via a page 3 location $0326. The address contained in $0326 (low-byte) and $0327 (high-byte) is the actual starting address of CHROUT, normally $F1CA.

This apparently roundabout method of calling subroutines is not peculiar to the Commodore 64. It is, in fact, a well-known solution to a problem which arises when an up-dated operating system is brought out. If operating system subroutines were called directly rather than using kernal jump vectors, they would, in all probability, be incorrect for an up-dated ROM unless the new designers tailored the software to maintain address compatibility, a restriction which would almost certainly degrade software efficiency. With kernal vectors, all that would be required is a rearrangement of the jump table in the RAM area. In fact the overall advantages are as follows:

(a) It allows the user freedom to *intercept* the standard operating system call by simply changing the vectored address.
(b) It allows the user to write extra code to *modify* the normal call before the original routine is entered.
(c) Operating syste ROMs can be updated or modified *without affecting previously written software.* All that needs to be changed by the new operating system is the vector contents.

Important kernal subroutines

As previously stated, Appendix B lists most of the subroutines together with the calling addresses and relevant details. However, some of the most commonly used examples are worth special treatment.

CHROUT

Function: to output a character at the next printing position to the currently opened output channel. If no other channel (such as the printer)

has been opened, the subroutine considers the channel to be the screen; it defaults to Device 6 which is the screen.
Calling address: $FFD2 or 64590 decimal.
Register involved: the ASCII code of the required character must be in the accumulator before calling CHROUT.

You are reminded that using JSR CHROUT (which is the preferred form) will only work if a previous assignment to $FFD2 or 64590 decimal has been made to the label CHROUT. If not, the absolute address form, JSR $FFD2 must be used. This reminder also applies to other subroutine calls.

CHRIN
Function: obtains a character from the currently open input channel and passes it to the accumulator. If no channel has previously been opened, the subroutine defaults to the keyboard. In addition to the above function, the cursor blink is turned on until CR (carriage return) is keyed in. The input from the keyboard is stored in the BASIC buffer which can hold up to 88 characters.
Calling address: $FFCF or 65487 decimal.
Register involved: accumulator.

The following example shows how to input a stream of characters from the keyboard and store sequentially in a block of memory. The process ends on receipt of a carriage return (ASCII 13).

```
        LDY #0
INPUT   JSR CHRIN
        STA BLOCK,Y
        INY
        CMP #13
        BNE INPUT
```

SCNKEY
Function: scan the keyboard for key pressed. If key is then pressed, the ASCII value is placed in the keyboard buffer.
Calling address: $FF9F or 65439.
Register involved: none.

GETIN
Function: Get a character from the currently open channel, defaulting to

the keyboard. Removes a character from the keyboard buffer and places its ASCII code in the accumulator.
Calling address: $FFE4 or 65508.
Register involved: accumulator.

The following example illustrates some of the subroutines described above:

```
TYPE   JSR SCNKEY
       JSR GETIN
       CMP #42
       BEQ FINISH
       JSR CHROUT
       JMP TYPE
FINISH RTS
```

SCNKEY puts the keyed character into the keyboard buffer. GETIN transfers the character from the buffer into the accumulator. CHROUT transfers the character from the accumulator to the screen. The sequence is terminated whenever an asterisk (ASCII 42) is entered. Provided the usual assignments are added to the above example, it can be used for typing practice.

Adding an array of integers

To consolidate the concept of building bricks, some of them are put together into a simple practical program that will sum an integer array.

Program 5.3 adds two-byte integer numbers held in a BASIC array (A%). For testing purposes only, the BASIC routine, Program 5.4, fills an array with random integers of mixed sign, the number of integers being specified by the user. The flowchart shown in Fig. 5.2 is given first in the hope it will help in the understanding of the listing.

Program 5.3 is the first one in this book which illustrates the speed of machine code. When assessing the speed, it should be realised that the filling of the array and the scrolled display of the numbers is carried out in BASIC. The speed referred to applies only to the machine code portion which performs the actual addition. A sequential addition check is carried out in BASIC, primarily for speed comparisons. To compare machine code speed with the BASIC equivalent, run the test program with 4000 integers. It will be seen that the machine code sum appears almost instantaneously after the numbers stop scrolling. The BASIC check on the addition takes many seconds. The program should be fairly easy to follow when traced in conjunction with the flowchart. It uses some of the coding blocks discussed in the earlier part of this chapter. For those without an assembler, the

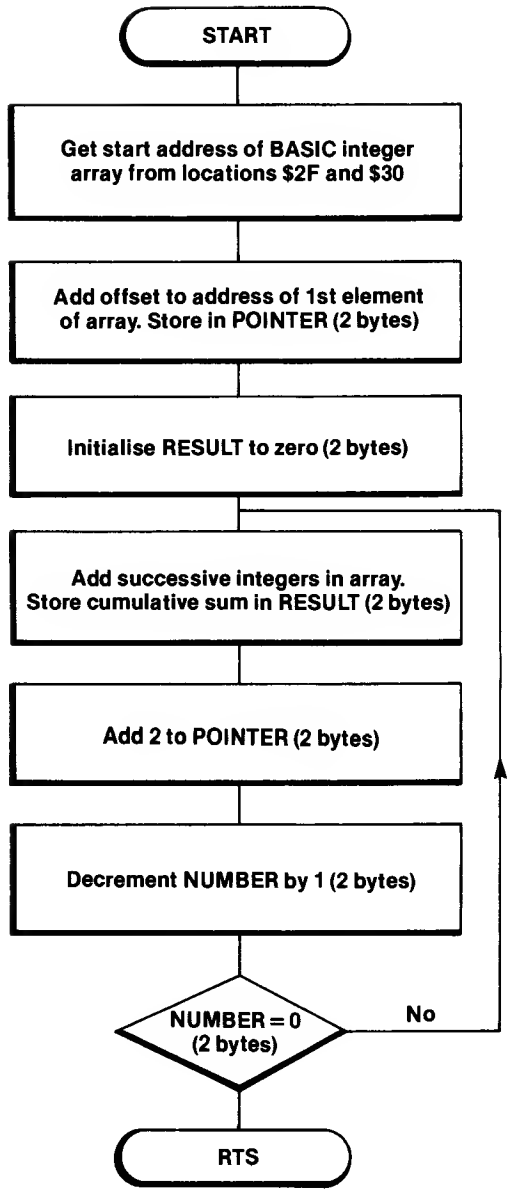


Fig. 5.2. Flowchart of Program 5.3.

```
10 ; SUMMATION OF AN INTEGER ARRAY
20 NUMBER       =  $FB
30 RESULT       =  $FD
40 POINTER      =  $4E
50 *=$C000
```

```
 60              CLC
 70              LDA $2F
 80              ADC #9
 90              STA POINTER
100              LDA $30
110              ADC #0
120              STA POINTER+1
130              LDA #0
140              STA RESULT
150              STA RESULT+1
160 LOOP         LDY #1
170              CLC
180              LDA (POINTER),Y
190              ADC RESULT
200              STA RESULT
210              DEY
220              LDA (POINTER),Y
230              ADC RESULT+1
240              STA RESULT+1
250              CLC
260              LDA POINTER
270              ADC #2
280              STA POINTER
290              BCC SKIP
300              INC POINTER+1
310 SKIP         SEC
320              LDA NUMBER
330              SBC #1
340              STA NUMBER
350              BCS SKIP2
360              DEC NUMBER+1
370              LDA NUMBER
380 SKIP2        BNE LOOP
390              LDA NUMBER+1
400              BNE LOOP
410              RTS
```

Program 5.3. Integer array summation.

```
10 REM INT ARRAY SUMMATION TEST PROGRAM
20 INPUT"HOW MANY INTEGERS";B%
30 REM FILL AND DISPLAY RANDOM ARRAY
40 DIM A%(B%)
50 FOR N=1 TO B%
60 A%(N)=1000*RND(1)
```

```
70 A%(N)=A%(N)-500
80 PRINT A%(N)
90 NEXT
100 REM SET UP NUMBER PARAMETER
110 HB%=B%/256
120 LB%=B%-(HB%*256)
130 REM PASS NUMBER PARAMETER
140 POKE 251,LB%
150 POKE 252,HB%
160 REM CALL SUMMATION ROUTINE
170 SYS 49152
180 PRINT
190 PRINT"SUMMATION M/CODE"
200 PRINT"= "PEEK(253)+PEEK(254)*256
210 PRINT
220 S%=0
230 FOR N=1 TO B%
240 S%=S%+A%(N)
250 NEXT
260 PRINT"CHECK USING BASIC="S%
```

Program 5.4. BASIC test program for integer array summation.

```
C000 18 A5 2F 69 09 85 4E A5
C008 30 69 00 85 4F A9 00 85
C010 FD 85 FE A0 01 18 B1 4E
C018 65 FD 85 FD 88 B1 4E 65
C020 FE 85 FE 18 A5 4E 69 02
C028 85 4E 90 02 E6 4F 38 A5
C030 FB E9 01 85 FB B0 02 C6
C038 FC A5 FB D0 D6 A5 FC D0
C040 D2 60
```

Hex Dump 5.1. Object code for Program 5.3.

object code of Program 5.3 is given in Hex Dump 5.1. This can be poked into memory with the hex loader routine given in Chapter 4 (Program 4.1).

Summary

1. Certain machine code patterns tend to reappear frequently and may be thought of as 'building bricks'.
2. Although building bricks are useful during development, some lines may be redundant when fitted into real programs.

3. The building brick habit could begin with machine code versions of structures encountered in BASIC.
4. It is unlikely that programs using only single byte numbers would have wide practical value.
5. The maximum signed numbers possible in a single byte are +127 and −128.
6. The maximum signed numbers possible in double byte representation are +32765 and −32766.
7. It is useful to name the two bytes representing one number as 'name' and 'name+1' – for example, NUMBER and NUMBER+1 for low- and high-byte respectively.
8. Remember to use CLC before adding low-byte pairs.
9. Remember to use SEC before subtracting low-byte pairs.
10. Never use CLC or SEC when handling high-byte pairs.
11. To multiply by 2, use ASL on low-byte and then ROL on high-byte each time.
12. To multiply by non-integral powers of 2, employ a mixture of shifting and adding.
13. Loops which count down towards zero are more efficient, but more error-prone, than those which count up.
14. There are useful subroutines already within the kernal which can be called up. They are defined in Appendix B, together with the calling addresses.
15. The most commonly used kernal subroutines will be CHROUT, CHRIN,SCNKEY and GETIN.

Self test

5.1 Write your own machine code routines to simulate the following BASIC lines:
(a) FOR N = 3 TO 21 STEP 3
(b) IF K = A−3 THEN J=J−1
(c) PRINT"GRANADA TECHNICAL BOOKS"

5.2 Write a routine to clear a double byte number held in NUMBER and NUMBER+1.

Chapter Six
Sort Routines

Introduction

Apart from personal interest and/or intellectual stimulation, there is little point in adopting a partisan approach to machine code. It is pointless to view BASIC as a language inferior to machine code. The two should complement, rather than rival, each other. Once familiarity and confidence is gained in handling machine code, it will gradually become clear which parts of a BASIC program should be relegated to machine code and which parts can be handled quite adequately in BASIC. There can be no doubt that one area in data processing, calling out for machine code solutions, is sorting data into numerical or alphabetical order. It has been stated that approximately 30% of all commercial computing time is spent on some kind of sorting activity. An ordered system of any kind represents a 'high energy' system. Since the equation for energy in physics is power multiplied by time, we would therefore expect that programs which sort data will make heavy demands on computing time.

The physical power of a given computer is fixed by the hardware, which in turn depends on such things as the clock frequency, wordlength and the sophistication built into the central processor (in the Commodore 64, the central processor is the 6510A). Although in no way meant as criticism, the machine, and indeed most other microcomputers likely to be found in the average home, are slow in terms of *mips* (millions of instructions per second). The Commodore 64 is rated at about 0.5 mips. In contrast, some of the modern mainframe giants have reached a speed approaching 100 mips with a wordlength of 64 bits and it is confidently expected that this figure will be substantially beaten by the forthcoming breed of fifth generation machines. Returning to present day reality, there is nothing we can do about the limitations imposed by the hardware of our machine. The only method of attack is by the use of software which takes the fullest advantage of the machine.

This chapter is devoted entirely to the problem of machine code sorting routines wich will be found useful in any programs designed to handle large amounts of data. In the home, for example, lists of names, addresses and telephone numbers are more valuable as information sources if arranged in

some sort of order. Stock control, payrolls and customer accounts in business are nearly always arranged in alphabetical or numerical order.

The programs in this chapter cover the sorting of integers, strings, floating point numbers and two-dimensional string arrays. They will also include routines which sort fixed length multi-field records. Programs are given, complete with a BASIC test program, so they can be entered and exhaustively examined. It should be pointed out that the machine code portion of the listings will stand alone as subroutines as long as (a) the correct parameters are passed from any BASIC program, and (b) the code is lodged either in one of the safe areas (not necessarily the areas used in our listings) or dynamically, above or below BASIC.

Bubble sort of a BASIC integer array

The bubble sort is well known but often despised because it is slow. It is one of the simplest sort routines to understand. However, providing there are not too many elements in the array, even bubble sorts are acceptable if written in machine code. Because the programs which follow are intended to be used in conjunction with BASIC, it is important to understand how the interpreter allocates variable space.

How integer array variables are stored

The two bytes, allocated to each integer array variable, are arranged as shown in Fig. 6.1.

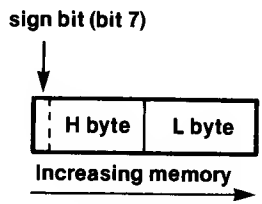


Fig. 6.1. How integers are stored.

Each integer of the array is stored sequentially so, in effect, each element has an address two bytes in advance of the previous integer.

The array header

The array itself always has a header. This is automatically set up in the form laid out in Fig. 6.2.

The first two bytes are the array name and correspond to the maximum of two distinguishable characters. For integer arrays, the first byte is the character ASCII code + $40. The second byte is the character ASCII code + $40, or just $40 if no second character exists in the array name. For

example, the array name A% would have $81 for the first byte ($41+$40) and $40 for the second since there is no second character.

The second two bytes are a pointer to the start of the next array stored in memory (if any). If the start address of the present array is incremented by the contents of these two locations then the address of the next array is obtained. Incidentally, this facility is used in the array search routine at the end of the chapter. The bytes mentioned above are followed by a single byte set to the number of dimensions in the array. The final two bytes of the header are the array size low-byte and high-byte respectively. The elements of the array, that is the actual values, are then stored sequentially in memory following the header in two-byte blocks, starting with the zeroth element. It is always useful to leave the zeroth element clear in many applications, so incrementing the array start address by 9 will give the address of the first array integer. A further increment of 2 will give the second, etc., and is the method used later in Program 6.1. However, if the zeroth element in the sort is to be included, an initial offset of 7 must be used.

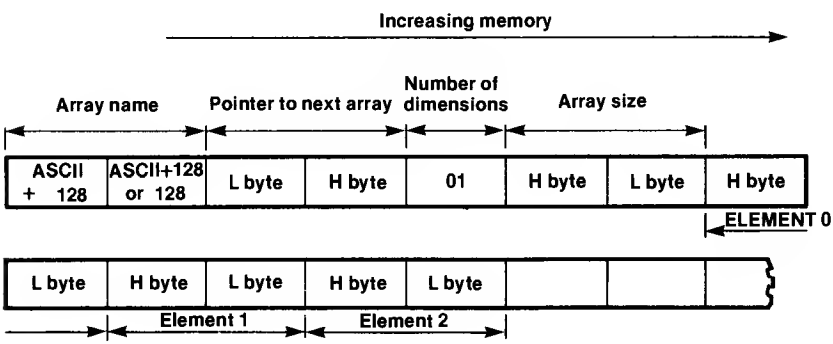


Fig. 6.2. How integer arrays are stored.

When a machine code routine is called from BASIC via SYS, it is necessary to POKE certain locations prior to the call in order to supply the routine with the essential parameters. The short program below shows how the parameters can be passed to sort routines before the call is made:

```
10 HB%=N%/256
20 LB%=N%-(HB%*256)
30 POKE 251,LB%
40 POKE 252,HB%
50 SYS 49152
```

The number of integers within the array is stored in N%. The BASIC integer variable N% will need to be split into its two component bytes for poking into locations 251 and 252. It is these locations which are used by

the sort routine to contain the number of integers. The high-byte HB% is formed by simply dividing by 256. Since HB% is an integer variable the INT function will occur automatically. The low-byte LB% is formed from the remainder by the equation shown in line 20. If the machine code is assembled at $C000 then SYS49152 will call the routine. Of course, the above variable names are arbitrarily chosen but they must be in the above order.

The flowchart of the bubble sort is given in Figure 6.3. As can be seen, the algorithm consists basically of an inner control loop and an outer control loop. The pairs of integers are repeatedly incremented, compared (and if necessary swopped) in the inner loop. The largest integer in the list always 'bubbles' through to the last position. It is no longer necessary to involve this integer again so the outer loop count may be reduced by one. On subsequent inner loop series of cycles, the next largest integer bubbles through to the last but one position in the list, and so on, until the list is fully sorted. The maximum number of comparisons is approximately equal to half the square of the array size.

The use of a swop flag often speeds up the execution of a bubble sort. The reason is that the program can exit early when no swops have been made during the current cycle. (This condition cannot be sensed by a standard bubble sort.) The savings are particularly noticeable when the array is only moderately disordered. In view of this extra addition to the bubble sort, the algorithm can be considered a bubble/exchange sort hybrid. Note from the flowchart that the blocks have been numbered for reference purposes. Block 7 is expanded in Fig. 6.4. The listing corresponding to the previous flowcharts (Figs. 6.3 and 6.4) is given in Program 6.1.

Breakdown of Program 6.1

Lines 30–70 set up the zero-page labelled locations referred to in assembly code.

Lines 90 to 140 decrement NUMBER (stored in two successive bytes) by one. NUMBER and NUMBER+1 are set up by poking locations 251 and 252 from BASIC. They contain the number of integers in the array low-byte and high-byte respectively. Note, that zero-page locations must be used for indirect indexed addressing.

Lines 150 to 210 store the address of the first element of the array in POINTER2 (2 bytes). This is done by picking up the start address of the array from locations $2F and $30 and adding the 9 offset as described earlier.

Lines 200 to 250 initialise the swop FLAG and CYCLE counter to zero.

Lines 260 to 290 copy POINTER2 (2 bytes) contents into POINTER1 (2 bytes).

Lines 300 to 340 adds 2 to POINTER1 and stores the result in POINTER2. The reason why 2 is added is so that POINTER2 is the address of the next integer in the array, that is to say, 2 bytes onwards.

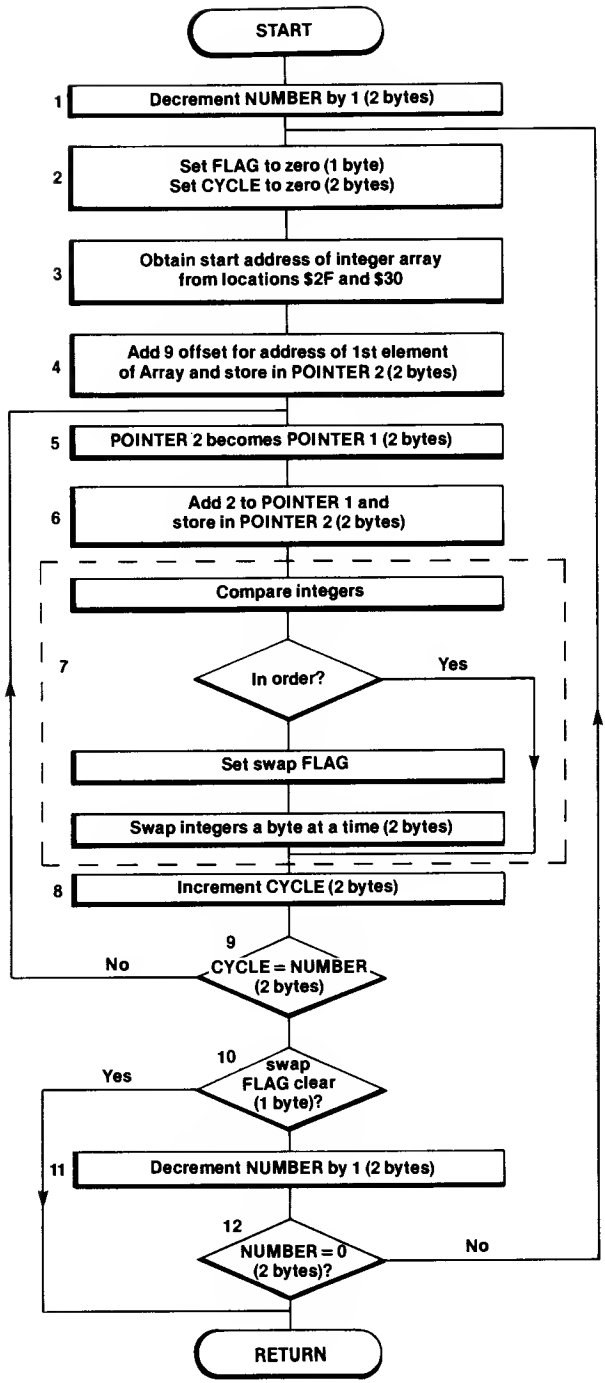


Fig. 6.3. Flowchart for integer array bubble sort.

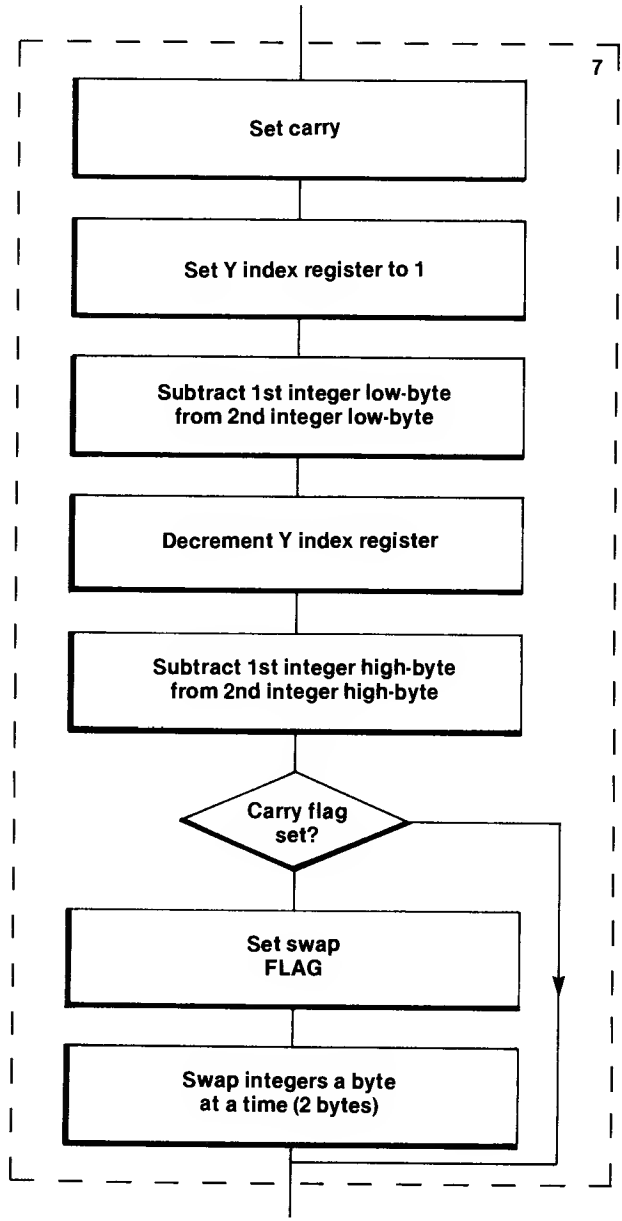


Fig. 6.4. Expansion of block 7.

```
10 ! BUBBLE SORT
20 ! ARRAY OF UNSIGNED INTEGERS
30 NUMBER        =  $FB
40 CYCLE         =  $FD
50 POINTER1      =  $57
```

```
 60 POINTER2      =  $59
 70 FLAG          =  $FF
 80 *=$C000
 90                SEC
100                LDA NUMBER
110                SBC #1
120                STA NUMBER
130                BCS OUTERLOOP
140                DEC NUMBER+1
150 OUTERLOOP      CLC
160                LDA $2F
170                ADC #9
180                STA POINTER2
190                LDA $30
200                ADC #0
210                STA POINTER2+1
220                LDA #0
230                STA FLAG
240                STA CYCLE
250                STA CYCLE+1
260 INNERLOOP      LDA POINTER2+1
270                STA POINTER1+1
280                LDA POINTER2
290                STA POINTER1
300                CLC
310                ADC #2
320                STA POINTER2
330                BCC SKIP
340                INC POINTER2+1
350 SKIP           LDY #1
360                SEC
370                LDA (POINTER2),Y
380                SBC (POINTER1),Y
390                DEY
400                LDA (POINTER2),Y
410                SBC (POINTER1),Y
420                BCS NOSWOP
430                INY
440                STY FLAG
450 SWOPLOOP       LDA (POINTER1),Y
460                TAX
470                LDA (POINTER2),Y
480                STA (POINTER1),Y
490                TXA
500                STA (POINTER2),Y
```

```
510                 DEY
520                 BPL  SWOPLOOP
530 NOSWOP          INC  CYCLE
540                 BNE  SKIP2
550                 INC  CYCLE+1
560 SKIP2           LDA  CYCLE
570                 CMP  NUMBER
580                 BNE  INNERLOOP
590                 LDA  CYCLE+1
600                 CMP  NUMBER+1
610                 BNE  INNERLOOP
620                 LDA  FLAG
630                 BEQ  FLAGCLEAR
640                 LDA  NUMBER
650                 SEC
660                 SBC  #1
670                 STA  NUMBER
680                 BCS  SKIP3
690                 DEC  NUMBER+1
700                 LDA  NUMBER
710 SKIP3           BNE  OUTERLOOP
720                 LDA  NUMBER+1
730                 BNE  OUTERLOOP
740 FLAGCLEAR       RTS
```

Program 6.1. Bubble sort of an unsigned integer array.

Line 350 initialises the Y index register to 1 for indirect indexed addressing. Lines 360 to 420 subtract the integers with carry, a byte at a time, keeping the result of the second bytes in the accumulator. Indirect indexed addressing is used so the Y register is decremented to pick up the high-byte. If the carry flag is set at the end of the subtraction, no swop is required. (This method only works for unsigned integers.)

Line 430 prepares the Y index register for the swop process. INY is used to set this to 1 since the current value of Y is 0.

Line 440 stores the Y register contents as a swop flag in FLAG (any non-zero quantity would do here).

Lines 450 to 520 swop the integers, a byte at a time, starting with the high byte. Notice that the X register is used as a temporary storage location because it is economical in terms of execution time. TAX uses only two machine cycles, whereas the alternatives PHA or STA require 3 cycles.

Lines 530 to 550 increment the CYCLE counter by 1. The coding given is economical in execution time.

Lines 560 to 610 compare the low-byte of CYCLE and NUMBER. If the result is non-zero, a branch is made to the label INNERLOOP. If the result

is zero, the program 'falls through' to compare the high-byte in the same manner.
Lines 620 to 630 checks if the swop FLAG is clear. If so, a branch to FLAGCLEAR is made.
Lines 640 to 690 decrement NUMBER by 1 (2 bytes of course).
Lines 700 to 730 check if NUMBER has reached zero, first checking the low-byte with branching to OUTERLOOP if not true, otherwise 'falling through' to compare the high-byte.

```
10 REM SORT TEST PROGRAM
20 REM ARRAY OF UNSIGNED INTEGERS
30 INPUT"SORT HOW MANY INTEGERS";B%
40 REM FILL AND DISPLAY RANDOM ARRAY
50 DIM A%(B%)
60 FOR N=1 TO B%
70 A%(N)=INT(32000*RND(1))
80 PRINT A%(N)
90 NEXT
100 PRINT"SORTING"
110 REM SET UP NUMBER PARAMETER
120 HB%=B%/256
130 LB%=B%-(HB%*256)
140 REM PASS NUMBER PARAMETER
150 POKE 251,LB%
160 POKE 252,HB%
170 REM CALL SORT ROUTINE
180 TI$="000000"
190 SYS 49152
200 T%=TI/60+0.5
210 REM DISPLAY SORTED ARRAY
220 FOR N=1 TO B%
230 PRINT A%(N)
240 NEXT
250 PRINT"SORTED"B%"INTEGERS IN"T%"SECONDS
```

Program 6.2. BASIC test program for unsigned integer array sort.

To test Program 6.1, a BASIC program on the lines of Program 6.2 can be used. This fills an integer array with random unsigned integers. The size of the array is selected by the user in line 30. The parameters are set up as previously described and the sort routine called with SYS49152. The sorted array is then displayed on the screen together with the time taken. It is not surprising that the time taken to set up and display the arrays often exceeds the time taken to sort them, due to the inherent slowness of BASIC. This is a good illustration of the speed of machine code.

```
C000 38 A5 FB E9 01 85 FB B0
C008 02 C6 FC 18 A5 2F 69 09
C010 85 59 A5 30 69 00 85 5A
C018 A9 00 85 FF 85 FD 85 FE
C020 A5 5A 85 58 A5 59 85 57
C028 18 69 02 85 59 90 02 E6
C030 5A A0 01 38 B1 59 F1 57
C038 88 B1 59 F1 57 B0 10 C8
C040 84 FF B1 57 AA B1 59 91
C048 57 8A 91 59 88 10 F3 E6
C050 FD D0 02 E6 FE A5 FD C5
C058 FB D0 C5 A5 FE C5 FC D0
C060 BF A5 FF F0 13 A5 FB 38
C068 E9 01 85 FB B0 04 C6 FC
C070 A5 FB D0 97 A5 FC D0 93
C078 60
```

Hex Dump 6.1. Object code for Program 6.1.

The hex dump is given to help those who have not yet purchased any form of assembler. The object code for Program 6.1 is given in Hex Dump 6.1. The code can be positioned from $C000 onwards by entering the bytes in the DATA section of the hex loader given and described in Chapter 4 (Program 4.1).

Bubble sort of a BASIC string array

This routine is capable of sorting a BASIC string array, where the string elements can vary in length up to the legal maximum of 255.

How string arrays are stored

When a string array is set up by the interpreter three bytes are used in a similar manner to integers but these bytes are not the strings themselves but the length and address details of where the string is actually stored. These three bytes can be referred to as a *string information block*, the details of which are given in Fig. 6.5.

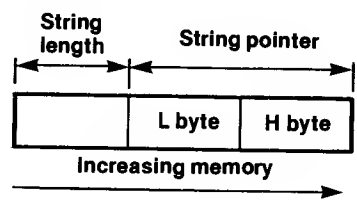


Fig. 6.5. How string information blocks are stored.

The actual string, consisting of the ASCII codes in sequential memory locations, is stored from the starting address given by the string pointer. A string array is formed by a series of such string information blocks, stored sequentially in memory. Therefore, if we want to swop strings (such as during a string sort) it is only necessary to swop the string information blocks since these tell the system where the strings are stored. This makes programs involving machine code sorting much easier because most of the work is already done by the interpreter. The string length byte of the string information block gives the actual length of the string in bytes (characters). A string array is stored in the format shown in Fig. 6.6.

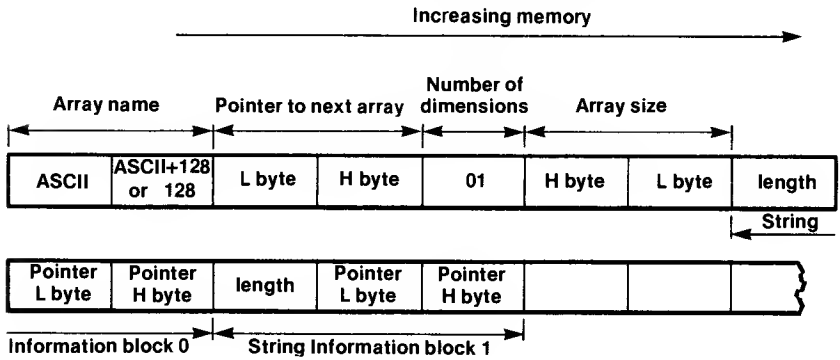


Fig. 6.6. How string arrays are stored.

With reference to Fig. 6.6, the header is much the same as that used for integers. However, the first two bytes that hold the array name are set out in a different way. This is so that the system can differentiate between different array types. For a string array, the first byte of the name is simply the ASCII character code. The second byte of the name is set to the ASCII code of the second character + $40, or just $40 if the second allowable character is not used. For example, the array name A$ would have $41 (the ASCII code for 'A') as the first byte and $40 as the second byte.

Again, as with the integer array sort, it is necessary to POKE locations 251 and 252 with the number of strings in the array prior to calling the sort routine with a SYS49152 command. The flowchart is essentially the same as Fig. 6.3 with one proviso, the details of block 7. The flowchart, showing the amendment is given in Fig. 6.7. The corresponding listing is given in Program 6.3.

It is only necessary here to explain the differences between Program 6.3 and Program 6.1. In Program 6.3, POINTER1 and POINTER2 are the address pointers to the string information blocks of the pair of strings. A further level of indirect indexed addressing is necessary to pick up the actual string characters since the string information block supplies only the address of where the string is stored.

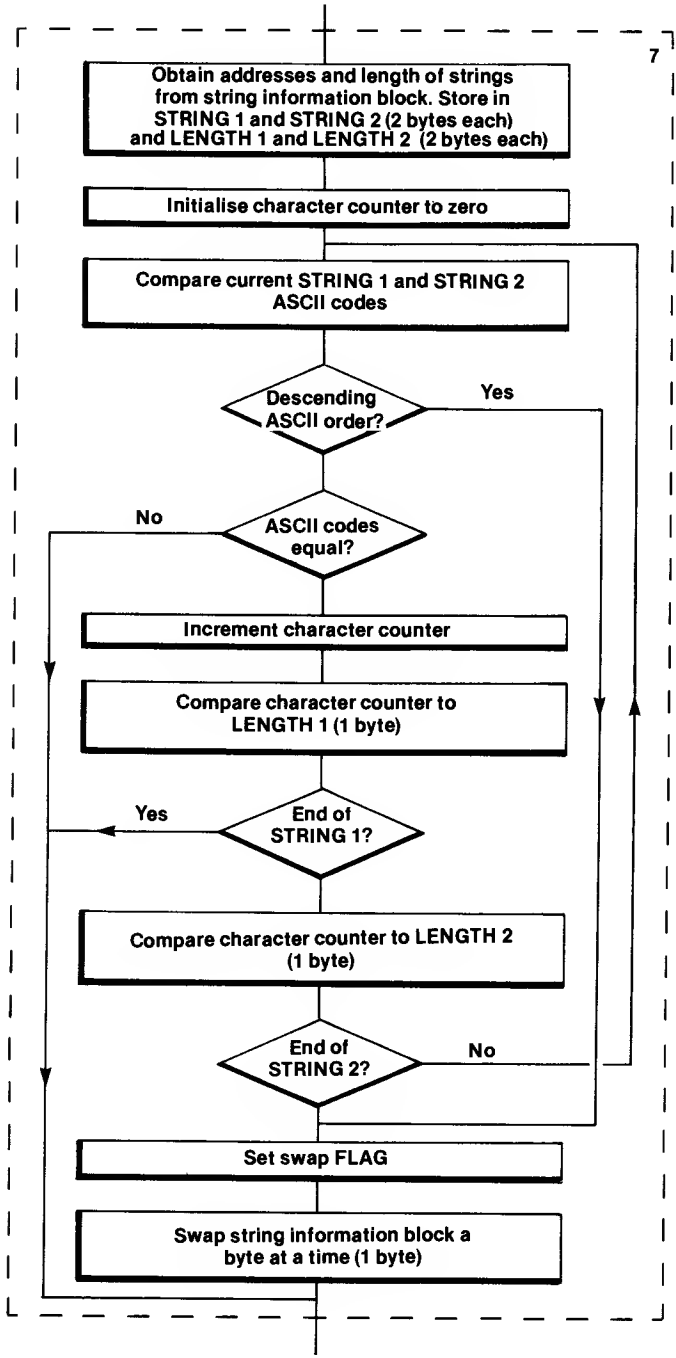


Fig. 6.7. Block 7 expansion for string array sort.

```
 10 !BUBBLE SORT
 20 !STRING ARRAY
 30 NUMBER        =  $FB
 40 CYCLE         =  $FD
 50 POINTER1      =  $57
 60 POINTER2      =  $59
 70 FLAG          =  $FF
 80 STRING1       =  $5B
 90 STRING2       =  $5D
100 LENGTH1       =  $5F
110 LENGTH2       =  $60
120 *=$C000
130               SEC
140               LDA NUMBER
150               SBC #1
160               STA NUMBER
170               BCS OUTERLOOP
180               DEC NUMBER+1
190 OUTERLOOP     CLC
200               LDA $2F
210               ADC #$0A
220               STA POINTER2
230               LDA $30
240               ADC #0
250               STA POINTER2+1
260               LDA #0
270               STA FLAG
280               STA CYCLE
290               STA CYCLE+1
300 INNERLOOP     LDA POINTER2+1
310               STA POINTER1+1
320               LDA POINTER2
330               STA POINTER1
340               CLC
350               ADC #3
360               STA POINTER2
370               BCC SKIP
380               INC POINTER2+1
390 SKIP          LDY #0
400               LDA (POINTER1),Y
410               STA LENGTH1
420               LDA (POINTER2),Y
430               STA LENGTH2
440               INY
450               LDA (POINTER1),Y
```

```
460                 STA STRING1
470                 LDA (POINTER2),Y
480                 STA STRING2
490                 INY
500                 LDA (POINTER1),Y
510                 STA STRING1+1
520                 LDA (POINTER2),Y
530                 STA STRING2+1
540                 LDY #0
550 COMPLOOP        LDA (STRING2),Y
560                 CMP (STRING1),Y
570                 BCC SWOP
580                 BNE NOSWOP
590                 INY
600                 CPY LENGTH1
610                 BEQ NOSWOP
620                 CPY LENGTH2
630                 BEQ SWOP
640                 BNE COMPLOOP
650 STAGE           BNE OUTERLOOP
660 SWOP            LDY #2
670                 STY FLAG
680 SWOPLOOP        LDA (POINTER1),Y
690                 TAX
700                 LDA (POINTER2),Y
710                 STA (POINTER1),Y
720                 TXA
730                 STA (POINTER2),Y
740                 DEY
750                 BPL SWOPLOOP
760 NOSWOP          INC CYCLE
770                 BNE SKIP2
780                 INC CYCLE+1
790 SKIP2           LDA CYCLE
800                 CMP NUMBER
810                 BNE INNERLOOP
820                 LDA CYCLE+1
830                 CMP NUMBER+1
840                 BNE INNERLOOP
850                 LDA FLAG
860                 BEQ FLAGCLEAR
870                 LDA NUMBER
880                 SEC
890                 SBC #1
900                 STA NUMBER
```

```
910                 BCS SKIP3
920                 DEC NUMBER+1
930                 LDA NUMBER
940 SKIP3           BNE STAGE
950                 LDA NUMBER+1
960                 BNE STAGE
970 FLAGCLEAR       RTS
```

Program 6.3. String array bubble sort.

Breakdown of Program 6.3

Lines 30 to 110 assign labels to frequently used locations.

Lines 130 to 180 decrement NUMBER (2 bytes) by one.

Lines 190 to 250 pick up the array start address from $2F and $30 and add $0A offset so as to point to the first element of the array. The result is then placed temporarily in the address pointer POINTER2 (2 bytes). The zeroth element is left clear for headings, etc.

Lines 260 to 290 initialise the locations labelled FLAG and CYCLE (2 bytes) to zero.

Lines 300 to 330 copy the contents of POINTER2 (2 bytes) to POINTER1 (2 bytes).

Lines 340 to 380 add 3, the length of a string information block, to POINTER2 (2 bytes).

Lines 390 to 410 use indirect indexed addressing to pick up the length of the first string from its string information block. This data is stored in LENGTH1 (1 byte).

Lines 420 to 430 pick up the length of the second string and store it in LENGTH2 (1 byte).

Lines 440 to 530 pick up the start addresses of the pair of strings, using indirect indexed addressing. The addresses are stored in zero page locations STRING1 (2 bytes) and STRING2 (2 bytes).

Line 540 sets the Y index register to zero. The Y register doubles as the string character counter.

Lines 550 to 580 compare the ASCII codes of the pair of strings picked up by indirect indexed addressing.

On comparison, as soon as the ASCII codes are found to be in descending order the strings are immediately swopped. If the ASCII codes are found to be ascending order, then no swop is required.

Line 590 increments the character counter.

Lines 600 to 610 compare the length of the first string (LENGTH1) to the character counter. If they are equal, a swop is not required.

Lines 620 to 630 compare the length of the second string (LENGTH2) to the character counter. If they are equal, a swop is forced.

Line 640 forces the branch to COMPLOOP which compares the next characters in the pair of strings, and so on.

Line 650 is an out-of-range branch patch. It is due to the displacement limit in relative addressing, which would have been exceeded in line 960. This method is often preferred to an absolute JMP instruction for object code relocation purposes.
Line 660 sets the Y register to 2. The Y register doubles as a byte counter and the index register for indirect indexed addressing.
Line 670 sets the swop flag. Any non-zero value can be stored in FLAG to indicate that a swop has occurred.
Lines 680 to 750 swop the 3-byte string information blocks, one byte at a time, using the X register as a temporary store.
Lines 760 to 970 are similar to lines 530 to 740 of Program 6.1.

The routine can be tested by the BASIC Program 6.4 which fills a string array with random strings of various lengths. The array size is entered by the user and split up into low-byte and high-byte components. These values are poked into locations 251 and 252 respectively and the sort routine called with SYS49152. The sorted string array is then displayed along with the sorting time. The object code is also supplied in the form of Hex Dump 6.2.

```
10 REM SORT TEST PROGRAM
20 REM STRING ARRAY
30 INPUT"SORT HOW MANY STRINGS";B%
40 REM FILL AND DISPLAY RANDOM ARRAY
50 DIM A$(B%)
60 FOR N=1 TO B%
70 B$=""
80 A%=10*RND(1)+1
90 FOR Z=1 TO A%
100 R%=26*RND(1)
110 K$=CHR$(R%+65)
120 B$=B$+K$
130 NEXT
140 A$(N)=B$
150 PRINT A$(N)
160 NEXT
170 PRINT:PRINT
180 PRINT"SORTING"
190 PRINT:PRINT
200 REM SET UP NUMBER PARAMETER
210 HB%=B%/256
220 LB%=B%-(HB%*256)
230 REM PASS NUMBER PARAMETER
240 POKE 251,LB%
250 POKE 252,HB%
260 REM CALL SORT ROUTINE
```

```
270 TI$="000000"
280 SYS 49152
290 T%=TI/60+0.5
300 REM DISPLAY SORTED ARRAY
310 FOR N=1 TO B%
320 PRINT A$(N)
330 NEXT
340 PRINT
350 PRINT"SORTED"B%"STRINGS IN"T%"SECONDS"
```

Program 6.4. BASIC test program for string array sort routines.

```
C000 38 A5 FB E9 01 85 FB B0
C008 02 C6 FC 18 A5 2F 69 0A
C010 85 59 A5 30 69 00 85 5A
C018 A9 00 85 FF 85 FD 85 FE
C020 A5 5A 85 58 A5 59 85 57
C028 18 69 03 85 59 90 02 E6
C030 5A A0 00 B1 57 85 5F B1
C038 59 85 60 C8 B1 57 85 5B
C040 B1 59 85 5D C8 B1 57 85
C048 5C B1 59 85 5E A0 00 B1
C050 5D D1 5B 90 0F D0 1E C8
C058 C4 5F F0 19 C4 60 F0 04
C060 D0 ED D0 A7 A0 02 84 FF
C068 B1 57 AA B1 59 91 57 8A
C070 91 59 88 10 F3 E6 FD D0
C078 02 E6 FE A5 FD C5 FB D0
C080 9F A5 FE C5 FC D0 99 A5
C088 FF F0 13 A5 FB 38 E9 01
C090 85 FB B0 04 C6 FC A5 FB
C098 D0 C8 A5 FC D0 C4 60
```

Hex Dump 6.2. Object code for Program 6.3.

Merge sort of BASIC integer array

Although the bubble sort routines given earlier are fast for small numbers of elements, the execution time increases alarmingly when in excess of about a hundred elements. To see the delay on high numbers, try running Program 6.1 with 1000 integers. You could well wait a minute or so before the sort was completed. A far better solution is to use a merge sort algorithm. We noted earlier that the bubble sort is fairly efficient if only a

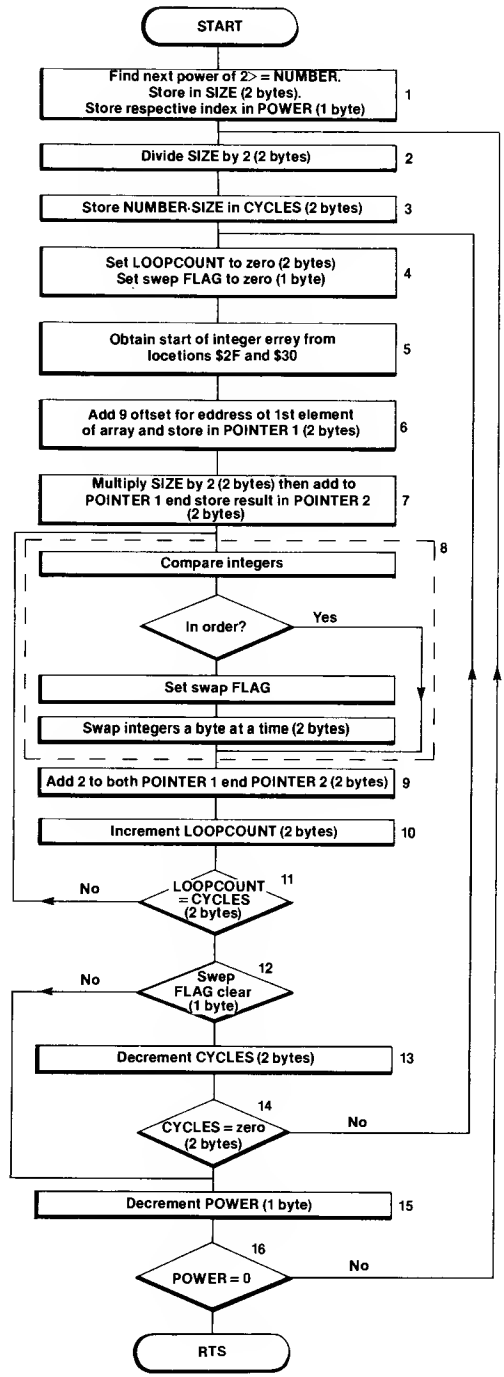


Fig. 6.8. Merge sort of an integer array.

small number of elements are to be sorted. We also noted that the use of a swop flag system significantly speeds up the execution of a bubble sort of a roughly ordered list. The merge sort algorithm takes advantage of both these virtues. Essentially, the array to be sorted is split up into small sets which are bubble sorted. These are merged to form larger sets which will be roughly in order. These larger sets are bubble sorted and merged to form even larger sets and so on until we are left with one large, roughly ordered list. This is finally bubble sorted which will be efficient due to the points made earlier. A flow-chart for a version of a merge sort is given in Fig. 6.8.

There is a danger of becoming intoxicated with verbosity in an attempt to explain the intricate details of a merge sort. A better grasp of the principles can be obtained by a *trace table*. In fact, any program which is difficult to follow will benefit from such an analysis. The idea is to follow the program through with arbitrary test data, keeping track of what happens to the various 'key' locations such as loop counters, etc. A trace table for the flowchart is given in Fig. 6.9. The unsorted array uses 8 random integers and shows how they would be sorted at various stages of

TRACE TABLE FOR A MERGE SORT

LOCATION LABELS	Value after 1st outer loop completed	Value after 2nd outer loop completed	Value after 3rd outer loop completed
NUMBER	8	8	8
POWER	2	1	0
SIZE	4	2	1
CYCLES	4	6	7

Unsorted array	Array after 1st outer loop completed	Array after 2nd outer loop completed	Array after 3rd outer loop completed
5	5	3	1
8	6	1	2
3	3	4	3
1	1	2	4
7	7	5	5
6	8	6	6
4	4	7	7
2	2	8	8

Fig. 6.9. Simple trace table of merge sort algorithm.

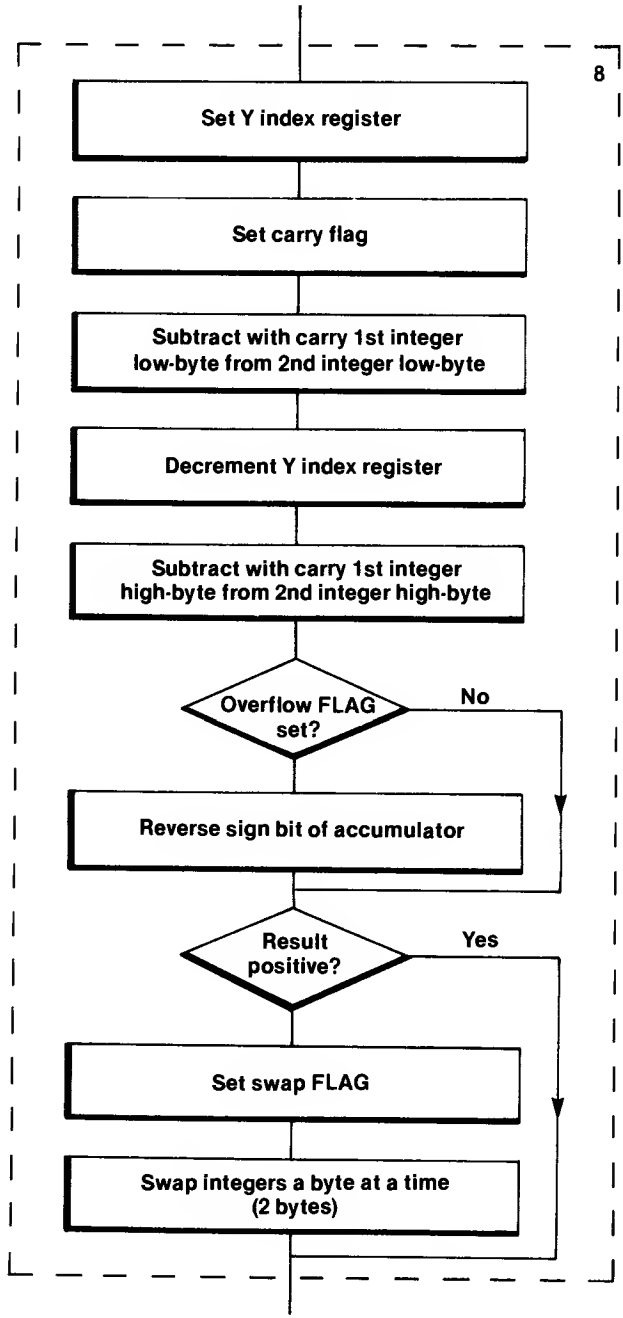


Fig. 6.10. Expansion of Block 8.

the trace. Notice how the array becomes more and more ordered as each outer loop cycle is completed.

If the flowchart and trace table have been understood, Program 6.5 should be reasonably easy to decipher. In this program, there is provision for sorting signed integers. An expansion of Block 8 of the flowchart is given in Fig. 6.10, showing the extra details required.

Program 6.5 assumes that the number of integers in the array are stored in locations 251 (low-byte) and 252 (high-byte). This is done using the POKE statement in BASIC prior to calling the routine with SYS49152. See Program 6.6 for details of how this is performed.

```
 10 !MERGE SORT
 20 !ARRAY OF SIGNED INTEGERS
 30 NUMBER        =  $FB
 40 CYCLES        =  $FD
 50 SIZE          =  $57
 60 STORE         =  $59
 70 LOOPCOUNT     =  $5A
 80 POWER         =  $5C
 90 POINTER1      =  $5D
100 POINTER2      =  $4E
110 FLAG          =  $FF
120 *=$C000
130               LDA #1
140               STA SIZE
150               LDA #0
160               STA POWER
170               STA SIZE+1
180 SIZELOOP      INC POWER
190               ASL SIZE
200               ROL SIZE+1
210               SEC
220               LDA SIZE
230               SBC NUMBER
240               LDA SIZE+1
250               SBC NUMBER+1
260               BCC SIZELOOP
270 OUTERLOOP     LSR SIZE+1
280               ROR SIZE
290               SEC
300               LDA NUMBER
310               SBC SIZE
320               STA CYCLES
330               LDA NUMBER+1
340               SBC SIZE+1
```

```
350                 STA CYCLES+1
360 MIDLOOP         LDA #0
370                 STA FLAG
380                 STA LOOPCOUNT
390                 STA LOOPCOUNT+1
400                 CLC
410                 LDA $2F
420                 ADC #9
430                 STA POINTER1
440                 LDA $30
450                 ADC #0
460                 STA POINTER1+1
470                 LDA SIZE+1
480                 STA STORE
490                 LDA SIZE
500                 ASL A
510                 ROL STORE
520                 CLC
530                 ADC POINTER1
540                 STA POINTER2
550                 LDA POINTER1+1
560                 ADC STORE
570                 STA POINTER2+1
580 INNERLOOP       LDY #1
590                 SEC
600                 LDA (POINTER2),Y
610                 SBC (POINTER1),Y
620                 DEY
630                 LDA (POINTER2),Y
640                 SBC (POINTER1),Y
650                 BVC NOOVERFLOW
660                 EOR #$80
670 NOOVERFLOW      BPL NOSWOP
680                 INY
690                 STY FLAG
700 SWOPLOOP        LDA (POINTER1),Y
710                 TAX
720                 LDA (POINTER2),Y
730                 STA (POINTER1),Y
740                 TXA
750                 STA (POINTER2),Y
760                 DEY
770                 BPL SWOPLOOP
780                 BMI NOSWOP
790 STAGE1          BNE MIDLOOP
```

```
 800 STAGE2       BNE OUTERLOOP
 810 NOSWOP       INC LOOPCOUNT
 820              BNE SKIP
 830              INC LOOPCOUNT+1
 840 SKIP         LDA POINTER1
 850              CLC
 860              ADC #2
 870              STA POINTER1
 880              BCC SKIP2
 890              INC POINTER1+1
 900 SKIP2        CLC
 910              LDA POINTER2
 920              ADC #2
 930              STA POINTER2
 940              BCC SKIP3
 950              INC POINTER2+1
 960 SKIP3        LDA CYCLES
 970              CMP LOOPCOUNT
 980              BNE INNERLOOP
 990              LDA CYCLES+1
1000              CMP LOOPCOUNT+1
1010              BNE INNERLOOP
1020              LDA FLAG
1030              BEQ FLAGCLEAR
1040              SEC
1050              LDA CYCLES
1060              SBC #1
1070              STA CYCLES
1080              BCS SKIP4
1090              DEC CYCLES+1
1100              LDA CYCLES
1110 SKIP4        BNE STAGE1
1120              LDA CYCLES+1
1130              BNE STAGE1
1140 FLAGCLEAR    DEC POWER
1150              BNE STAGE2
1160              RTS
```

Program 6.5. Merge sort of an array of signed integers.

Breakdown of Program 6.5

Lines 30 to 110 assign labels to often used locations.
Line 120 causes assembly at location $C000.

Lines 90 to 120 obtain the address of the BASIC variable NUMBER % from the CALL parameter block. The address is placed in zero-page locations STORE (2 bytes).
Lines 130 to 170 initialise the locations POWER (1 byte) and SIZE (2 bytes).
Lines 180 to 260 are a block of code dedicated to finding the next power of 2, greater than or equal to the contents of NUMBER (2 bytes). The result is stored in SIZE (2 bytes). The corresponding power index is stored in POWER (1 byte). The op-codes ASL and ROL in conjunction are convenient for 2-byte manipulation of powers of 2.
Lines 270 to 280 divide SIZE by 2 by shifting right.
Lines 290 to 350 subtract SIZE from NUMBER and store the result in CYCLES (2 bytes).
Lines 360 to 390 initialise LOOPCOUNT (2 bytes) and the swop FLAG (1 byte) to zero.
Lines 400 to 460 pick up the address of the first element in the array by adding 9 to the contents of locations $2F (low-byte) and $30 (high-byte). These locations hold the address of the start of array space and 9 is added to account for the array header bytes. The result is stored in POINTER1 (2 bytes).
Lines 470 to 570 are a block of code which multiplies SIZE (2 bytes) by 2 and adds POINTER1 (2 bytes), storing the result in POINTER 2 (2 bytes). The reason why it is multiplied by 2 is because each integer occupies two bytes. The multiplication is achieved by shifting left once.
Lines 580 to 640 subtract the first integer from the second, low-byte then high-byte, keeping the most significant byte of the result in the accumulator (this has the most important sign bit). Indirect indexed addressing is used to pick up the integers from memory. Remember that integers are stored in memory with the high-byte lowest in memory.
Line 650 checks if the V flag is set. If clear, it skips line 660.
Line 660 assumes that the V flag is set so reverses the sign bit.
Line 670 tests the sign of the accumulator contents and bypasses the swop loop if positive (including zero). This ensures that if both integers are the same, no swopping occurs.
Lines 680 and 690 set the Y index register to 1 again. This is also a convenient place to set the swop flag so that the Y register contents are stored in FLAG. Any non-zero value would have done equally well.
Lines 700 to 770 swop the integers, a byte at a time, using indirect indexed addressing.
Line 780 serves no useful purpose in the program, other than causing a bypass of the out of range branch-patch section (lines 790 to 800). This is due to the relative branch limit being exceeded later on in the program. A staging post is needed to branch back again. An absolute jump could have been used but the advantage of relocating the object code would have been lost. This may be a problem for those without assemblers.

Lines 810 to 830 increment LOOPCOUNT (2 bytes).
Lines 840 to 950 add the usual 4 to each of POINTER1 (2 bytes) and POINTER2 (2 bytes).
Lines 960 to 1010 compare LOOPCOUNT to CYCLES, branching to INNERLOOP if not equal.
Lines 1020 to 1030 test the swop flag and, if clear, branch to FLAGCLEAR.
Lines 1040 to 1090 decrement CYCLES (2 bytes).
Lines 1100 to 1130 compare CYCLES to zero, branching to MIDLOOP via STAGE1, the out of range branch-patch.
Lines 1140 to 1150 decrement POWER (1 byte) and compare it to zero, branching if not zero to OUTERLOOP via STAGE2.

```
10 REM SORT TEST PROGRAM
20 REM ARRAY OF SIGNED INTEGERS
30 INPUT"SORT HOW MANY INTEGERS";B%
40 REM FILL AND DISPLAY RANDOM ARRAY
50 DIM A%(B%)
60 FOR N=1 TO B%
70 A%(N)=INT(32000*RND(1))
80 A%(N)=A%(N)-16000
90 PRINT A%(N)
100 NEXT
110 PRINT"SORTING"
120 REM SET UP NUMBER PARAMETER
130 HB%=B%/256
140 LB%=B%-(HB%*256)
150 REM PASS NUMBER PARAMETER
160 POKE 251,LB%
170 POKE 252,HB%
180 REM CALL SORT ROUTINE
190 TI$="000000"
200 SYS 49152
210 T%=TI/60+0.5
220 REM DISPLAY SORTED ARRAY
230 FOR N=1 TO B%
240 PRINT A%(N)
250 NEXT
260 PRINT"SORTED"B%"INTEGERS IN"T%"SECONDS
```

Program 6.6. BASIC program for testing signed integer array sort routine.

Program 6.6 is a simple BASIC testing program which sets up an array of signed integers, passes the usual parameters and calls the sort routine with SYS49152. Finally the sorted array is displayed along with the sorting time.

For those without assemblers the object code of Program 6.6 is given in Hex Dump 6.3.

```
C000 A9 01 85 57 A9 00 85 5C
C008 85 58 E6 5C 06 57 26 58
C010 38 A5 57 E5 FB A5 58 E5
C018 FC 90 EF 46 58 66 57 38
C020 A5 FB E5 57 85 FD A5 FC
C028 E5 58 85 FE A9 00 85 FF
C030 85 5A 85 5B 18 A5 2F 69
C038 09 85 5D A5 30 69 00 85
C040 5E A5 58 85 59 A5 57 0A
C048 26 59 18 65 5D 85 4E A5
C050 5E 65 59 85 4F A0 01 38
C058 B1 4E F1 5D 88 B1 4E F1
C060 5D 50 02 49 80 10 16 C8
C068 84 FF B1 5D AA B1 4E 91
C070 5D 8A 91 4E 88 10 F3 30
C078 04 D0 B1 D0 9E E6 5A D0
C080 02 E6 5B A5 5D 18 69 02
C088 85 5D 90 02 E6 5E 18 A5
C090 4E 69 02 85 4E 90 02 E6
C098 4F A5 FD C5 5A D0 B6 A5
C0A0 FE C5 5B D0 B0 A5 FF F0
C0A8 13 38 A5 FD E9 01 85 FD
C0B0 B0 04 C6 FE A5 FD D0 C1
C0B8 A5 FE D0 BD C6 5C D0 BB
C0C0 60
```

Hex Dump 6.3. Object code for Program 6.6.

Merge sort of a BASIC string array

The overall structure of the next program is similar to Program 6.5, the main difference being the substitution of the string comparison segment for block 8 in the flowchart. This segment has been described in detail for the bubble sort. For other minor differences, refer to the program breakdown section.

Since the overall structure is similar to that of Program 6.5 the breakdown of Program 6.7 will only stress the differences.

```
10 !MERGE SORT
20 !STRING ARRAY
```

```
 30 NUMBER        =  $FB
 40 CYCLES        =  $FD
 50 POINTER1      =  $57
 60 POINTER2      =  $59
 70 FLAG          =  $FF
 80 STRING1       =  $5B
 90 STRING2       =  $5D
100 LENGTH1       =  $5F
110 LENGTH2       =  $60
120 POWER         =  $4E
130 SIZE          =  $4F
140 LOOPCOUNT     =  $51
150 STORE         =  $26
160 *=$C000
170               LDA #1
180               STA SIZE
190               LDA #0
200               STA POWER
210               STA SIZE+1
220 SIZELOOP      INC POWER
230               ASL SIZE
240               ROL SIZE+1
250               SEC
260               LDA SIZE
270               SBC NUMBER
280               LDA SIZE+1
290               SBC NUMBER+1
300               BCC SIZELOOP
310 OUTERLOOP     LSR SIZE+1
320               ROR SIZE
330               SEC
340               LDA NUMBER
350               SBC SIZE
360               STA CYCLES
370               LDA NUMBER+1
380               SBC SIZE+1
390               STA CYCLES+1
400 MIDLOOP       LDA #0
410               STA FLAG
420               STA LOOPCOUNT
430               STA LOOPCOUNT+1
440               CLC
450               LDA $2F
460               ADC #$0A
470               STA POINTER1
```

```
480                 LDA $30
490                 ADC #0
500                 STA POINTER1+1
510                 LDA SIZE+1
520                 STA STORE
530                 LDA SIZE
540                 ASL A
550                 ROL STORE
560                 CLC
570                 ADC SIZE
580                 ADC POINTER1
590                 STA POINTER2
600                 LDA STORE
610                 ADC SIZE+1
620                 ADC POINTER1+1
630                 STA POINTER2+1
640 INNERLOOP       LDY #0
650                 LDA (POINTER1),Y
660                 STA LENGTH1
670                 LDA (POINTER2),Y
680                 STA LENGTH2
690                 INY
700                 LDA (POINTER1),Y
710                 STA STRING1
720                 LDA (POINTER2),Y
730                 STA STRING2
740                 INY
750                 LDA (POINTER1),Y
760                 STA STRING1+1
770                 LDA (POINTER2),Y
780                 STA STRING2+1
790                 LDY #0
800 COMPLOOP        LDA (STRING2),Y
810                 CMP (STRING1),Y
820                 BCC SWOP
830                 BNE NOSWOP
840                 INY
850                 CPY LENGTH1
860                 BEQ NOSWOP
870                 CPY LENGTH2
880                 BEQ SWOP
890                 BNE COMPLOOP
900 STAGE1          BNE MIDLOOP
910 STAGE2          BNE OUTERLOOP
```

```
 920 SWOP          LDY #2
 930               STY FLAG
 940 SWOPLOOP      LDA (POINTER1),Y
 950               TAX
 960               LDA (POINTER2),Y
 970               STA (POINTER1),Y
 980               TXA
 990               STA (POINTER2),Y
1000               DEY
1010               BPL SWOPLOOP
1020 NOSWOP        INC LOOPCOUNT
1030               BNE SKIP
1040               INC LOOPCOUNT+1
1050 SKIP          LDA POINTER1
1060               CLC
1070               ADC #3
1080               STA POINTER1
1090               BCC SKIP2
1100               INC POINTER1+1
1110 SKIP2         LDA POINTER2
1120               CLC
1130               ADC #3
1140               STA POINTER2
1150               BCC SKIP3
1160               INC POINTER2+1
1170 SKIP3         LDA CYCLES
1180               CMP LOOPCOUNT
1190               BNE INNERLOOP
1200               LDA CYCLES+1
1210               CMP LOOPCOUNT+1
1220               BNE INNERLOOP
1230               LDA FLAG
1240               BEQ FLAGCLEAR
1250               SEC
1260               LDA CYCLES
1270               SBC #1
1280               STA CYCLES
1290               BCS SKIP4
1300               DEC CYCLES+1
1310               LDA CYCLES
1320 SKIP4         BNE STAGE1
1330               LDA CYCLES+1
1340               BNE STAGE1
1350 FLAGCLEAR     DEC POWER
```

```
1360                    BNE STAGE2
1370                    RTS
```

Program 6.7. Merge sort of a BASIC string array.

Breakdown of Program 6.7

Lines 80 to 110 assign labels to the extra locations needed.

Lines 440 to 500 add $A offset instead of 9 to the array start address to point to the first element of the array. This is because string information blocks require an extra three locations for each element.

Lines 510 to 620 multiply SIZE (2 bytes) by 3. This is also to accommodate the extra length of the string information block. The process is carried out by shifting SIZE left once and adding in the original SIZE contents. The contents of POINTER1 is then added and the result stored in POINTER2. In lines 1060 to 1160, three is added to both POINTER1 and POINTER2 for the same reason.

The routine can be called and tested with the same BASIC test program as that used for the bubble sort. The object code of Program 6.7 is given in Hex Dump 6.4.

```
C000 A9 01 85 4F A9 00 85 4E
C008 85 50 E6 4E 06 4F 26 50
C010 38 A5 4F E5 FB A5 50 E5
C018 FC 90 EF 46 50 66 4F 38
C020 A5 FB E5 4F 85 FD A5 FC
C028 E5 50 85 FE A9 00 85 FF
C030 85 51 85 52 18 A5 2F 69
C038 0A 85 57 A5 30 69 00 85
C040 58 A5 50 85 26 A5 4F 0A
C048 26 26 18 65 4F 65 57 85
C050 59 A5 26 65 50 65 58 85
C058 5A A0 00 B1 57 85 5F B1
C060 59 85 60 C8 B1 57 85 5B
C068 B1 59 85 5D C8 B1 57 85
C070 5C B1 59 85 5E A0 00 B1
C078 5D D1 5B 90 11 D0 20 C8
C080 C4 5F F0 1B C4 60 F0 06
C088 D0 ED D0 A0 D0 8D A0 02
C090 84 FF B1 57 AA B1 59 91
C098 57 8A 91 59 88 10 F3 E6
C0A0 51 D0 02 E6 52 A5 57 18
C0A8 69 03 85 57 90 02 E6 58
C0B0 A5 59 18 69 03 85 59 90
```

```
C0B8 02 E6 5A A5 FD C5 51 D0
C0C0 98 A5 FE C5 52 D0 92 A5
C0C8 FF F0 13 38 A5 FD E9 01
C0D0 85 FD B0 02 C6 FE A5 FD
C0D8 D0 B0 A5 FE D0 AC C6 4E
C0E0 D0 AA 60
```

Hex Dump 6.4. Object code for Program 6.7.

Merge sort of an unsigned BASIC floating point array

Programs which handle records or tables often need to sort unsigned floating point numbers (most numerical values in such programs are unsigned). It is therefore important to have at least an outline understanding of how floating point numbers are stored in the Commodore 64.

A floating point number consists of a *mantissa* and an *exponent*. Four bytes are allocated to the mantissa and one for the exponent. The most significant bit of the exponent is the sign bit and is in *reverse two's complement*. That is to say, a negative exponent has '0' as the sign bit, and a '1' indicates a positive exponent. The reason for this rather strange practice is that the maximum possible negative exponent closely approaches zero. This means that zero can be loosely taken as the most negative exponent. Therefore less negative exponents, through to positive, correspond to a progression of increasingly larger exponents.

From a mathematical viewpoint, a mantissa is always positive so no sign bit is required. Therefore, the sign bit in the mantissa is used to denote the sign of the entire number in conventional two's complement form.

How floating point numbers are stored

Floating point numbers are stored in a five-byte form, the details of which are given in Figure 6.11. When an array of floating point numbers is set up,

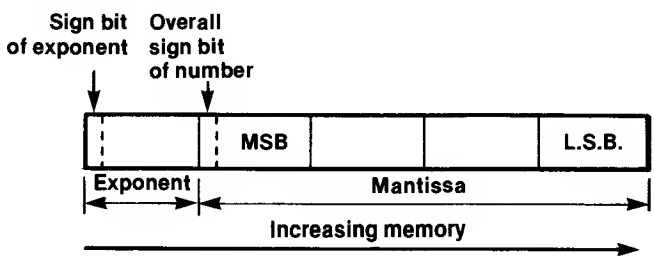


Fig. 6.11. How floating point numbers are stored.

a header is used in a similar manner to other types of array described previously. The details are shown in Fig. 6.12. The only difference, other than the method of storing each element, is the contents of the first two bytes which name the array. The first byte is the ASCII code of the first

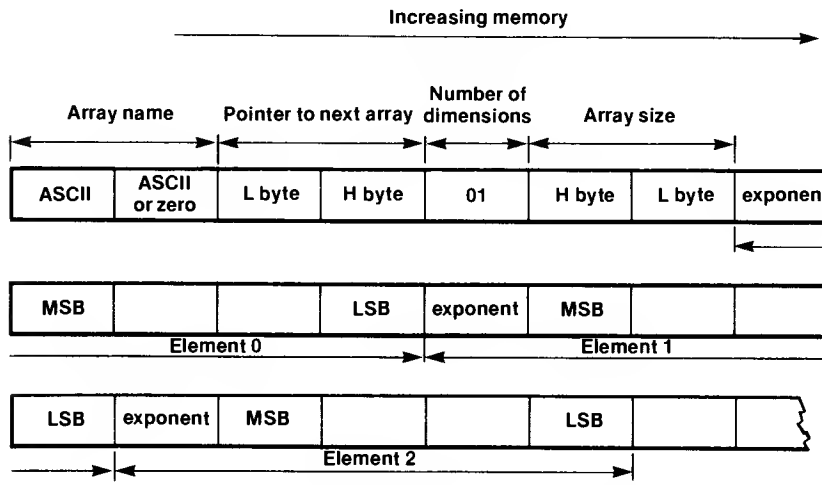


Fig. 6.12. How a floating point array is stored.

character and the second byte is the ASCII code of the second character of the array name. The second byte is set to zero if the second distinguishable character is not used. For example, if the array is named A, the first byte will be set to $41 and the second set to zero. Another point worth noticing is that the first element is offset from the start of the array by $0C bytes. However, if the zeroth element is to be used the offset is reduced to 8 bytes.

The overall structure of the flowchart differs from that of Fig. 6.8, mainly in Block 8. The element comparison block is replaced by Fig. 6.13. There are other minor differences due to the increased number of bytes required for storing each array element. The breakdown of Program 6.8 will detail the differences only. It should also be remembered that the number of array elements must be poked into the locations 251 and 252 as before. The complete listing is shown in Program 6.8.

Breakdown of Program 6.8

Lines 400 to 460 arrange for an offset of $C to be added to the array start address in order to set POINTER1 (2 bytes) to the first element in the array. Lines 460 to 610 multiply SIZE (2 bytes) by five to account for the increased number of bytes used to store each element. This is done by shifting SIZE left twice and adding in the original value of SIZE. The contents of

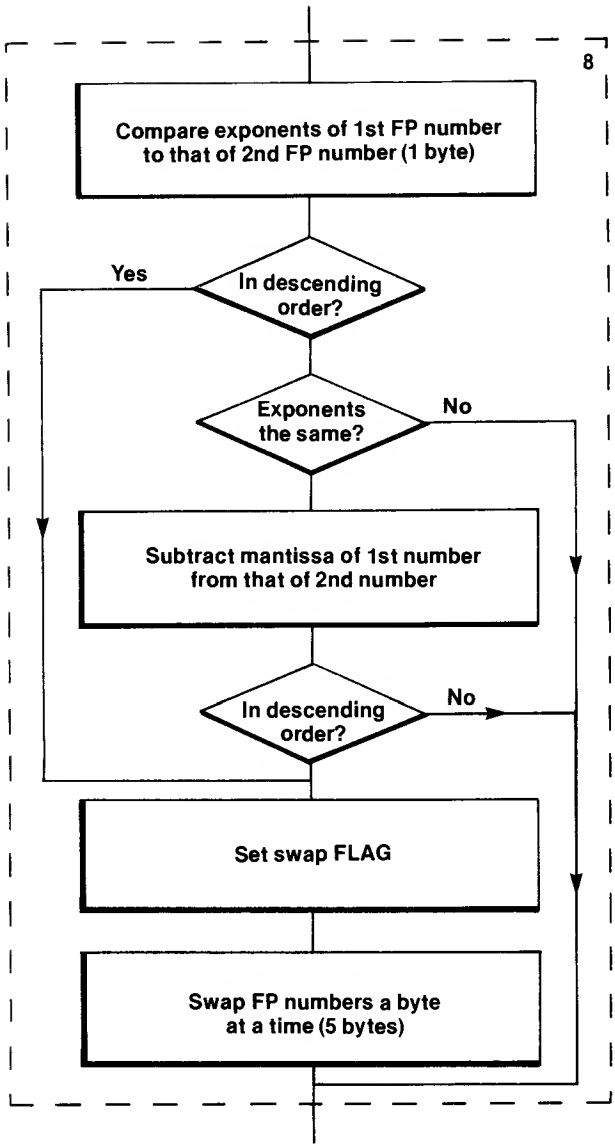


Fig. 6.13. Block 8 expansion for unsigned floating merge sort.

POINTER1 (2 bytes) are added and the result stored in POINTER2 (2 bytes).
Line 620 initialises the Y index register for indirect indexed addressing.
Lines 630 to 660 compare the exponent bytes of the accessed pair of

```
 10 ! MERGE SORT
 20 ! ARRAY OF UNSIGNED FP NUMBERS
 30 NUMBER        =  $FB
 40 CYCLES        =  $FD
 50 SIZE          =  $57
 60 STORE         =  $59
 70 LOOPCOUNT     =  $5B
 80 POWER         =  $5D
 90 POINTER1      =  $5E
100 POINTER2      =  $4E
110 FLAG          =  $FF
120 *=$C000
130               LDA #1
140               STA SIZE
150               LDA #0
160               STA POWER
170               STA SIZE+1
180 SIZELOOP      INC POWER
190               ASL SIZE
200               ROL SIZE+1
210               SEC
220               LDA SIZE
230               SBC NUMBER
240               LDA SIZE+1
250               SBC NUMBER+1
260               BCC SIZELOOP
270 OUTERLOOP     LSR SIZE+1
280               ROR SIZE
290               SEC
300               LDA NUMBER
310               SBC SIZE
320               STA CYCLES
330               LDA NUMBER+1
340               SBC SIZE+1
350               STA CYCLES+1
360 MIDLOOP       LDA #0
370               STA FLAG
380               STA LOOPCOUNT
390               STA LOOPCOUNT+1
400               CLC
410               LDA $2F
420               ADC #$0C
430               STA POINTER1
440               LDA $30
450               ADC #0
```

```
460                 STA POINTER1+1
470                 LDA SIZE+1
480                 STA STORE+1
490                 LDA SIZE
500                 ASL A
510                 ROL STORE+1
520                 ASL A
530                 ROL STORE+1
540                 CLC
550                 ADC SIZE
560                 ADC POINTER1
570                 STA POINTER2
580                 LDA STORE+1
590                 ADC SIZE+1
600                 ADC POINTER1+1
610                 STA POINTER2+1
620 INNERLOOP       LDY #0
630                 LDA (POINTER2),Y
640                 CMP (POINTER1),Y
650                 BCC SWOP
660                 BNE NOSWOP
670                 LDY #4
680                 SEC
690 COMPLOOP        LDA (POINTER2),Y
700                 SBC (POINTER1),Y
710                 DEY
720                 BNE COMPLOOP
730                 BCS NOSWOP
740 SWOP            LDY #4
750                 STY FLAG
760 SWOPLOOP        LDA (POINTER1),Y
770                 TAX
780                 LDA (POINTER2),Y
790                 STA (POINTER1),Y
800                 TXA
810                 STA (POINTER2),Y
820                 DEY
830                 BPL SWOPLOOP
840                 BMI NOSWOP
850 STAGE1          BNE MIDLOOP
860 STAGE2          BNE OUTERLOOP
870 NOSWOP          INC LOOPCOUNT
880                 BNE SKIP
890                 INC LOOPCOUNT+1
900 SKIP            LDA POINTER1
```

```
 910                CLC
 920                ADC #5
 930                STA POINTER1
 940                BCC SKIP2
 950                INC POINTER1+1
 960 SKIP2          CLC
 970                LDA POINTER2
 980                ADC #5
 990                STA POINTER2
1000                BCC SKIP3
1010                INC POINTER2+1
1020 SKIP3          LDA CYCLES
1030                CMP LOOPCOUNT
1040                BNE INNERLOOP
1050                LDA CYCLES+1
1060                CMP LOOPCOUNT+1
1070                BNE INNERLOOP
1080                LDA FLAG
1090                BEQ FLAGCLEAR
1100                SEC
1110                LDA CYCLES
1120                SBC #1
1130                STA CYCLES
1140                BCS SKIP4
1150                DEC CYCLES+1
1160                LDA CYCLES
1170 SKIP4          BNE STAGE1
1180                LDA CYCLES+1
1190                BNE STAGE1
1200 FLAGCLEAR      DEC POWER
1210                BNE STAGE2
1220                RTS
```

Program 6.8. Merge sort of an unsigned floating point array.

floating point array elements. The element corresponding to POINTER1 is considered as the first of the pair. If they are in descending order, the elements are swopped. If already in ascending order, the elements are not swopped. If the exponents are equal the program falls through to line 670. Lines 670 to 730 subtract the mantissas a byte at a time keeping the result of the most significant bytes in the accumulator. If the carry flag is clear at the end, indicating descending order, the elements are swopped. If the carry flag is set the swop routine is bypassed. This includes the case where the mantissas are equal.

Lines 850 to 860 are out-of-range branch patches used for reasons outlined earlier.

Lines 900 to 1010 add 5 to POINTER1 rather than 2 or 3 as in the previous programs. This is due to the increased byte length of the array elements (5 bytes each).

The routine can be tested by means of Program 6.9, and is similar to the previous test programs, apart from setting up the relevant type of array. The object code of Program 6.8 is given in Hex Dump 6.5. The hex dump can be placed into memory with the help of the hex loader (Program 4.1) given in Chapter 4.

```
10 REM SORT TEST PROGRAM
20 REM FLOATING POINT ARRAY
30 INPUT"SORT HOW MANY FP NUMBERS";B%
40 REM FILL AND DISPLAY RANDOM ARRAY
50 DIM A(B%)
60 FOR N=1 TO B%
70 A(N)=1500*RND(1)
80 PRINT A(N)
90 NEXT
100 PRINT"SORTING"
110 REM SET UP NUMBER PARAMETER
120 HB%=B%/256
130 LB%=B%-(HB%*256)
140 REM PASS NUMBER PARAMETER
150 POKE 251,LB%
160 POKE 252,HB%
170 REM CALL SORT ROUTINE
180 TI$="000000"
190 SYS 49152
200 T%=TI/60+0.5
210 REM DISPLAY SORTED ARRAY
220 FOR N=1 TO B%
230 PRINT A(N)
240 NEXT
250 PRINT"SORTED"B%"FP NUMBERS IN"T%"SECONDS'
```

Program 6.9. BASIC test program for floating point array sort routines.

Machine code sort routines applied to BASIC multifield filing programs

There are two common methods of generating multifield records in BASIC. One is to use *fields of fixed length* substrings where the complete record is stored as one string array element. The other is to create a *two-dimensional string array* where the records occupy one dimension and the

```
C000 A9 01 85 57 A9 00 85 5D
C008 85 58 E6 5D 06 57 26 58
C010 38 A5 57 E5 FB A5 58 E5
C018 FC 90 EF 46 58 66 57 38
C020 A5 FB E5 57 85 FD A5 FC
C028 E5 58 85 FE A9 00 85 FF
C030 85 5B 85 5C 18 A5 2F 69
C038 0C 85 5E A5 30 69 00 85
C040 5F A5 58 85 5A A5 57 0A
C048 26 5A 0A 26 5A 18 65 57
C050 65 5E 85 4E A5 5A 65 58
C058 65 5F 85 4F A0 00 B1 4E
C060 D1 5E 90 0E D0 23 A0 04
C068 38 B1 4E F1 5E 88 D0 F9
C070 B0 17 A0 04 84 FF B1 5E
C078 AA B1 4E 91 5E 8A 91 4E
C080 88 10 F3 30 04 D0 A5 D0
C088 92 E6 5B D0 02 E6 5C A5
C090 5E 18 69 05 85 5E 90 02
C098 E6 5F 18 A5 4E 69 05 85
C0A0 4E 90 02 E6 4F A5 FD C5
C0A8 5B D0 B1 A5 FE C5 5C D0
C0B0 AB A5 FF F0 13 38 A5 FD
C0B8 E9 01 85 FD B0 02 C6 FE
C0C0 A5 FD D0 C1 A5 FE D0 BD
C0C8 C6 5D D0 BB 60
```

Hex Dump 6.5. Object code for Program 6.8.

fields occupy the other. This is often referred to as a row/column file format. Both have advantages and disadvantages. The former method is more economical when storing records since only one string information block is set up per record by the interpreter. On the other hand, BASIC programming can be tedious and expensive on memory. The latter method makes for concise programming in BASIC but is heavy on string information blocks (the number of fields multiplied by number of records). It is a matter of personal preference which method is used so a machine code merge sort routine to handle each type of record format will be given. The requirement of any routine of this type is to sort entire records according to any specified field, therefore additional calling parameters will be necessary.

Merge sort of fixed length multifield records

The complete source code listing is given in Program 6.10, the BASIC test program in Program 6.11, and the object code in Hex Dump 6.6. The

overall structure of Program 6.10 is similar to that of a simple string array merge sort but with a few extra complications due to the component field substrings.The extra coding and changes are detailed in the program breakdown. Two new zero-page locations are assigned labels. The first is FIELDST which stores the first character position of the field within the string array element. The second is FIELDEND, the last character position of the field within the string array element. By convention, the zero element is reserved for headings, labels, etc.

```
 10 ! MERGE SORT OF MULTIFIELD
 20 ! FIXED LENGTH RECORDS
 30 NUMBER        =  $FB
 40 FIELDST       =  $FD
 50 FIELDEND      =  $FE
 60 POINTER1      =  $57
 70 POINTER2      =  $59
 80 FLAG          =  $FF
 90 STRING1       =  $5B
100 STRING2       =  $5D
110 CYCLES        =  $5F
120 POWER         =  $4E
130 SIZE          =  $4F
140 LOOPCOUNT     =  $51
150 STORE         =  $26
160 *=$C000
170               LDA #1
180               STA SIZE
190               LDA #0
200               STA POWER
210               STA SIZE+1
220 SIZELOOP      INC POWER
230               ASL SIZE
240               ROL SIZE+1
250               SEC
260               LDA SIZE
270               SBC NUMBER
280               LDA SIZE+1
290               SBC NUMBER+1
300               BCC SIZELOOP
310 OUTERLOOP     LSR SIZE+1
320               ROR SIZE
330               SEC
340               LDA NUMBER
350               SBC SIZE
360               STA CYCLES
```

```
370                 LDA NUMBER+1
380                 SBC SIZE+1
390                 STA CYCLES+1
400 MIDLOOP         LDA #0
410                 STA FLAG
420                 STA LOOPCOUNT
430                 STA LOOPCOUNT+1
440                 CLC
450                 LDA $2F
460                 ADC #$0A
470                 STA POINTER1
480                 LDA $30
490                 ADC #0
500                 STA POINTER1+1
510                 LDA SIZE+1
520                 STA STORE
530                 LDA SIZE
540                 ASL A
550                 ROL STORE
560                 CLC
570                 ADC SIZE
580                 ADC POINTER1
590                 STA POINTER2
600                 LDA STORE
610                 ADC SIZE+1
620                 ADC POINTER1+1
630                 STA POINTER2+1
640 INNERLOOP       LDY #1
650                 LDA (POINTER1),Y
660                 STA STRING1
670                 LDA (POINTER2),Y
680                 STA STRING2
690                 INY
700                 LDA (POINTER1),Y
710                 STA STRING1+1
720                 LDA (POINTER2),Y
730                 STA STRING2+1
740                 LDY FIELDST
750                 DEY
760 COMPLOOP        LDA (STRING2),Y
770                 CMP (STRING1),Y
780                 BCC SWOP
790                 BNE NOSWOP
800                 INY
810                 CPY FIELDEND
```

```
 820                BNE COMPLOOP
 830                BEQ NOSWOP
 840 STAGE1         BNE MIDLOOP
 850 STAGE2         BNE OUTERLOOP
 860 SWOP           LDY #2
 870                STY FLAG
 880 SWOPLOOP       LDA (POINTER1),Y
 890                TAX
 900                LDA (POINTER2),Y
 910                STA (POINTER1),Y
 920                TXA
 930                STA (POINTER2),Y
 940                DEY
 950                BPL SWOPLOOP
 960 NOSWOP         INC LOOPCOUNT
 970                BNE SKIP
 980                INC LOOPCOUNT+1
 990 SKIP           LDA POINTER1
1000                CLC
1010                ADC #3
1020                STA POINTER1
1030                BCC SKIP2
1040                INC POINTER1+1
1050 SKIP2          LDA POINTER2
1060                CLC
1070                ADC #3
1080                STA POINTER2
1090                BCC SKIP3
1100                INC POINTER2+1
1110 SKIP3          LDA CYCLES
1120                CMP LOOPCOUNT
1130                BNE INNERLOOP
1140                LDA CYCLES+1
1150                CMP LOOPCOUNT+1
1160                BNE INNERLOOP
1170                LDA FLAG
1180                BEQ FLAGCLEAR
1190                SEC
1200                LDA CYCLES
1210                SBC #1
1220                STA CYCLES
1230                BCS SKIP4
1240                DEC CYCLES+1
1250                LDA CYCLES
1260 SKIP4          BNE STAGE1
```

```
1270                LDA CYCLES+1
1280                BNE STAGE1
1290 FLAGCLEAR      DEC POWER
1300                BNE STAGE2
1310                RTS
```

Program 6.10. Merge sort of multifield fixed-length records.

Breakdown of Program 6.10

Lines 40 to 50 assign labels to the extra locations needed.

Lines 740 to 820 swop the entire string (record) according to the field substring comparison. The Y index register is initially loaded with the contents of FIELDST. Using indirect indexed addressing, the substring (field) is then compared character by character, branching as necessary. The loop is concluded with a comparison of the contents of FIELDEND with the Y index register.

Program 6.11, the BASIC program used for testing, sets up a random string array of any specified length up to 255. These strings can be considered as the form eventually used for record storage. The number of records (elements) in the array are, as usual, poked into locations 251 and 252. The field-start (FS%) and field-end (FE%) character positions are poked into locations 253 and 254 respectively. The routine is called with SYS49152 and the sorted array displayed on the screen. On examination of the display the entire string element will be sorted according to a smaller substring or field as requested.

```
10 REM SORT TEST PROGRAM
20 REM MULTIFIELD FIXED LENGTH RECORDS
30 INPUT"SORT HOW MANY RECORDS";B%
35 INPUT"HOW MANY CHARACTERS EACH";V%
40 INPUT"FIELD START POSITION";FS%
50 INPUT"FIELD END POSITION  ";FE%
60 IF FS%>V% OR FE%>V% THEN 30
70 REM FILL AND DISPLAY RANDOM ARRAY
80 DIM A$(B%)
90 FOR N=1 TO B%
100 B$=""
110 FOR Z=1 TO V%
120 R%=26*RND(1)
130 K$=CHR$(R%+65)
140 B$=B$+K$
150 NEXT
160 A$(N)=B$
170 PRINT A$(N)
```

```
180 NEXT
190 PRINT:PRINT
200 PRINT"SORTING"
210 PRINT:PRINT
220 REM SET UP NUMBER PARAMETER
230 HB%=B%/256
240 LB%=B%-(HB%*256)
250 REM PASS PARAMETERS
260 POKE 251,LB%
270 POKE 252,HB%
280 POKE 253,FS%
290 POKE 254,FE%
300 REM CALL SORT ROUTINE
310 TI$="000000"
320 SYS 49152
330 T%=TI/60+0.5
340 REM DISPLAY SORTED ARRAY
350 FOR N=1 TO B%
360 PRINT A$(N)
370 NEXT
380 PRINT
390 PRINT"SORTED"B%"RECORDS IN"T%"SECONDS"
```

Program 6.11. BASIC test program for multifield fixed-length record sorts.

```
C000 A9 01 85 4F A9 00 85 4E
C008 85 50 E6 4E 06 4F 26 50
C010 38 A5 4F E5 FB A5 50 E5
C018 FC 90 EF 46 50 66 4F 38
C020 A5 FB E5 4F 85 5F A5 FC
C028 E5 50 85 60 A9 00 85 FF
C030 85 51 85 52 18 A5 2F 69
C038 0A 85 57 A5 30 69 00 85
C040 58 A5 50 85 26 A5 4F 0A
C048 26 26 18 65 4F 65 57 85
C050 59 A5 26 65 50 65 58 85
C058 5A A0 01 B1 57 85 5B B1
C060 59 85 5D C8 B1 57 85 5C
C068 B1 59 85 5E A4 FD 88 B1
C070 5D D1 5B 90 0D D0 1C C8
C078 C4 FE D0 F3 F0 15 D0 AC
C080 D0 99 A0 02 84 FF B1 57
C088 AA B1 59 91 57 8A 91 59
C090 88 10 F3 E6 51 D0 02 E6
C098 52 A5 57 18 69 03 85 57
```

```
C0A0 90 02 E6 58 A5 59 18 69
C0A8 03 85 59 90 02 E6 5A A5
C0B0 5F C5 51 D0 A4 A5 60 C5
C0B8 52 D0 9E A5 FF F0 13 38
C0C0 A5 5F E9 01 85 5F B0 02
C0C8 C6 60 A5 5F D0 B0 A5 60
C0D0 D0 AC C6 4E D0 AA 60
```

Hex Dump 6.6. Object code for Program 6.10.

Merge sort of a two-dimensional string array

The string information blocks, corresponding to multidimensional string array elements, are stored sequentially in memory. The following series shows the order in which they occur for a two-dimensional string array in the Commodore 64.

A$(0,0),A$(1,0),A$(2,0),A$(3,0),A$(0,1),A$(1,1).....A$(3,N).

The array is set up with a header, the locations corresponding with that shown in Fig. 6.14.

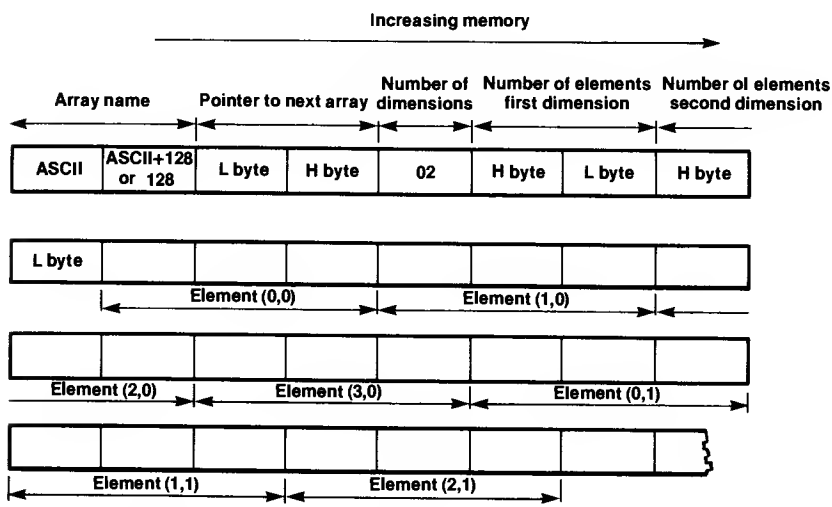


Fig. 6.14. How a two-dimensional string array is stored.

If a file is DIMensioned in BASIC as A$(NF%,NR%) then a file A$ can be considered as NR% records, each containing NF% fields. We can define a specific field of a specific record by A$(F,R) where F stands for field

number and R for record number. The parameters that need passing from BASIC to a sort routine are:

(a) the number of fields, NF%
(b) the number of records, NR%
(c) the field index number, F%.

These quantities are needed by the sort routine, Program 6.12, and can be poked directly into memory by the method shown in the BASIC test program, Program 6.13. Of these, only NR% may occupy two bytes. The number of fields are poked into the location labelled NUMBYTE. The number of records are poked into NUMBER and NUMBER+1. Finally the index number of the field by which the records are to be sorted is poked into OFFSET. The complete source code listing and BASIC test routine are shown in Programs 6.12 and 6.13 respectively.

Since each string information block occupies 3 bytes and there are NF%+1 fields to each record, the sort routine will need to calculate the number of bytes necessary before swopping the total string information blocks corresponding to each record. Referring to Program 6.12, this is performed by lines 350 to 400 and the result is overwritten in NUMBYTE (1 byte). This piece of code evaluates the expression: NUMBYTE=3*(NF%+1).

The by now familiar SI block address pointers POINTER1 and POINTER2 refer to the zeroth element of NF% so an offset needs to be calculated to point to the required sort-field element SI block in the NF% dimension (remember that 3 bytes offset is required for each). This is performed by lines 410 to 450 and the result is overwritten in OFFSET (1 byte). The segment of code calculates the expression OFFSET=3*(F%+1). By setting the Y register to OFFSET and using indirect indexed addressing, the required sort field SI block positions in the record can be accessed. The code responsible is at lines 910 to 1050. Plainly, in lines 740 to 900, NUMBYTE is used as the multiplier for SIZE, rather than 3 used in previous single dimension string array sorts. Another difference is that we will need to add NUMBYTE instead of 3 to the previous SI block address pointers in order to access the next record. This code is at lines 1330 to 1380. Apart from these differences, the overall structure is similar to that previously described.

The A$(F,0) elements are excluded from the sort since these elements are useful for column headings, labels, etc. in a practical filing system. The routine can sort records with up to a maximum of integer 128/3=42 fields (not much of a handicap in practice). Program 6.7 derivatives have been well-tried and tested in a practical filing system and will sort a computerful of records in a second or so.The object code for Program 6.12 is given in Hex Dump 6.7 and can be loaded directly into memory with the hex loader, Program 4.1. However, if you have an assembler it is better to enter the source code.

```
 10 !MERGE SORT OF A TWO DIMENSIONAL
 20 !STRING ARRAY (FIELD,RECORD) FORMAT
 30 NUMBER        =  $FB
 40 NUMBYTE       =  $FD
 50 OFFSET        =  $FE
 60 POINTER1      =  $57
 70 POINTER2      =  $59
 80 FLAG          =  $FF
 90 STRING1       =  $5B
100 STRING2       =  $5D
110 LENGTH1       =  $5F
120 LENGTH2       =  $60
130 START         =  $61
140 POWER         =  $4E
150 SIZE          =  $4F
160 LOOPCOUNT     =  $51
170 STORE         =  $26
180 CYCLES        =  $28
190 *=$C000
200               LDA #1
210               STA SIZE
220               LDA #0
230               STA POWER
240               STA SIZE+1
250 SIZELOOP      INC POWER
260               ASL SIZE
270               ROL SIZE+1
280               SEC
290               LDA SIZE
300               SBC NUMBER
310               LDA SIZE+1
320               SBC NUMBER+1
330               BCC SIZELOOP
340               LDA NUMBYTE
350               CLC
360               ADC #1
370               STA NUMBYTE
380               ASL A
390               ADC NUMBYTE
400               STA NUMBYTE
410               CLC
420               LDA OFFSET
430               ASL A
440               ADC OFFSET
450               STA OFFSET
```

```
460                 CLC
470                 LDA $2F
480                 ADC #9
490                 ADC NUMBYTE
500                 STA START
510                 LDA $30
520                 ADC #0
530                 STA START+1
540 OUTERLOOP       LSR SIZE+1
550                 ROR SIZE
560                 SEC
570                 LDA NUMBER
580                 SBC SIZE
590                 STA CYCLES
600                 LDA NUMBER+1
610                 SBC SIZE+1
620                 STA CYCLES+1
630 MIDLOOP         LDA #0
640                 STA FLAG
650                 STA LOOPCOUNT
660                 STA LOOPCOUNT+1
670                 LDA START
680                 STA POINTER1
690                 LDA START+1
700                 STA POINTER1+1
710                 LDA #0
720                 STA STORE
730                 STA STORE+1
740                 LDX NUMBYTE
750 MULTLOOP        CLC
760                 LDA STORE
770                 ADC SIZE
780                 STA STORE
790                 LDA STORE+1
800                 ADC SIZE+1
810                 STA STORE+1
820                 DEX
830                 BNE MULTLOOP
840                 CLC
850                 LDA POINTER1
860                 ADC STORE
870                 STA POINTER2
880                 LDA POINTER1+1
890                 ADC STORE+1
900                 STA POINTER2+1
```

```
 910 INNERLOOP   LDY OFFSET
 920             LDA (POINTER1),Y
 930             STA LENGTH1
 940             LDA (POINTER2),Y
 950             STA LENGTH2
 960             INY
 970             LDA (POINTER1),Y
 980             STA STRING1
 990             LDA (POINTER2),Y
1000             STA STRING2
1010             INY
1020             LDA (POINTER1),Y
1030             STA STRING1+1
1040             LDA (POINTER2),Y
1050             STA STRING2+1
1060             LDY #0
1070 COMPLOOP    LDA (STRING2),Y
1080             CMP (STRING1),Y
1090             BCC SWOP
1100             BNE NOSWOP
1110             INY
1120             CPY LENGTH1
1130             BEQ NOSWOP
1140             CPY LENGTH2
1150             BEQ SWOP
1160             BNE COMPLOOP
1170 STAGE1      BNE MIDLOOP
1180 STAGE2      BNE OUTERLOOP
1190 SWOP        LDY NUMBYTE
1200             DEY
1210             STY FLAG
1220 SWOPLOOP    LDA (POINTER1),Y
1230             TAX
1240             LDA (POINTER2),Y
1250             STA (POINTER1),Y
1260             TXA
1270             STA (POINTER2),Y
1280             DEY
1290             BPL SWOPLOOP
1300 NOSWOP      INC LOOPCOUNT
1310             BNE SKIP
1320             INC LOOPCOUNT+1
1330 SKIP        LDA POINTER1
1340             CLC
1350             ADC NUMBYTE
```

```
1360                STA POINTER1
1370                BCC SKIP2
1380                INC POINTER1+1
1390 SKIP2          LDA POINTER2
1400                CLC
1410                ADC NUMBYTE
1420                STA POINTER2
1430                BCC SKIP3
1440                INC POINTER2+1
1450 SKIP3          LDA CYCLES
1460                CMP LOOPCOUNT
1470                BNE INNERLOOP
1480                LDA CYCLES+1
1490                CMP LOOPCOUNT+1
1500                BNE INNERLOOP
1510                LDA FLAG
1520                BEQ FLAGCLEAR
1530                SEC
1540                LDA CYCLES
1550                SBC #1
1560                STA CYCLES
1570                BCS SKIP4
1580                DEC CYCLES+1
1590                LDA CYCLES
1600 SKIP4          BNE STAGE1
1610                LDA CYCLES+1
1620                BNE STAGE1
1630 FLAGCLEAR      DEC POWER
1640                BNE STAGE2
1650                RTS
```

Program 6.12. Merge sort of a two-dimensional string array sort.

```
10 REM SORT TEST PROGRAM
20 REM 2 DIMENSIONAL STRING ARRAY
30 INPUT"SORT HOW MANY RECORDS";NR%
40 INPUT"NUMBER OF FIELDS";NF%
50 INPUT"SORT WHICH FIELD";F%
60 REM FILL AND DISPLAY RANDOM ARRAY
70 DIM A$(NF%,NR%)
80 FOR R=1 TO NR%
90 FOR F=1 TO NF%
100 B$=""
110 A%=10*RND(1)+1
120 FOR Z=1 TO A%
```

```
130 R%=26*RND(1)
140 K$=CHR$(R%+65)
150 B$=B$+K$
160 NEXT
170 A$(F,R)=B$
180 PRINTA$(F,R)
190 NEXT
200 PRINT
210 NEXT
220 PRINT:PRINT
230 PRINT"SORTING"
240 PRINT:PRINT
250 REM SET UP NUMBER PARAMETER
260 HB%=NR%/256
270 LB%=NR%-(HB%*256)
280 REM PASS PARAMETERS
290 POKE 251,LB%
300 POKE 252,HB%
310 POKE 253,NF%
320 POKE 254,F%
330 REM CALL SORT ROUTINE
340 TI$="000000"
350 SYS 49152
360 T%=TI/60+0.5
370 REM DISPLAY SORTED ARRAY
380 FOR R=1 TO NR%
390 FOR F=1 TO NF%
400 PRINT A$(F,R)
410 NEXT
420 PRINT
430 NEXT
440 PRINT"SORTED"NR%"RECORDS IN"T%"SECONDS"
```

```
C000 A9 01 85 4F A9 00 85 4E
C008 85 50 E6 4E 06 4F 26 50
C010 38 A5 4F E5 FB A5 50 E5
C018 FC 90 EF A5 FD 18 69 01
C020 85 FD 0A 65 FD 85 FD 18
C028 A5 FE 0A 65 FE 85 FE 18
C030 A5 2F 69 09 65 FD 85 61
C038 A5 30 69 00 85 62 46 50
C040 66 4F 38 A5 FB E5 4F 85
C048 28 A5 FC E5 50 85 29 A9
```

```
C050 00 85 FF 85 51 85 52 A5
C058 61 85 57 A5 62 85 58 A9
C060 00 85 26 85 27 A6 FD 18
C068 A5 26 65 4F 85 26 A5 27
C070 65 50 85 27 CA D0 F0 18
C078 A5 57 65 26 85 59 A5 58
C080 65 27 85 5A A4 FE B1 57
C088 85 5F B1 59 85 60 C8 B1
C090 57 85 5B B1 59 85 5D C8
C098 B1 57 85 5C B1 59 85 5E
C0A0 A0 00 B1 5D D1 5B 90 11
C0A8 D0 21 C8 C4 5F F0 1C C4
C0B0 60 F0 06 D0 ED D0 98 D0
C0B8 85 A4 FD 88 84 FF B1 57
C0C0 AA B1 59 91 57 8A 91 59
C0C8 88 10 F3 E6 51 D0 02 E6
C0D0 52 A5 57 18 65 FD 85 57
C0D8 90 02 E6 58 A5 59 18 65
C0E0 FD 85 59 90 02 E6 5A A5
C0E8 28 C5 51 D0 97 A5 29 C5
C0F0 52 D0 91 A5 FF F0 13 38
C0F8 A5 28 E9 01 85 28 B0 02
C100 C6 29 A5 28 D0 AF A5 29
C108 D0 AB C6 4E D0 A9 60
```

Hex Dump 6.7. Object code for Program 6.12.

Searching for an array

All the programs in this chapter assume that the array to be sorted is the first stored in the array space. This may not be the case. In certain applications, many arrays could be stored so it is well to discuss the modifications necessary to search for a particular named array.

The address pointers $2F and $30 hold the address of the start of array space, low-byte and high-byte respectively. Referring to Fig. 6.2, this address is byte 1 of the array header corresponding to the first array stored in memory. By adding the contents of bytes 3 and 4 to this address, low and high bytes respectively, the next array address is calculated. The process can be repeated till the end of array space is reached, the pointer to the end of array space being $31 (low-byte) and $32 (high-byte).

The search problem is easily solved. All that is needed is to find the start of each array, as above, and test whether the codes of the array names satisfy the search requirement. This process is performed by Program 6.14. The specified array-name codes are placed into the locations 251 ($FB) and

```
 10 !START OF ARRAY SEARCH ROUTINE
 20 NAME          =  $FB
 30 ADDRESS       =  $FD
 40 OUTPUT        =  $FFD2
 50 *=$C000
 60               LDA $2F
 70               STA ADDRESS
 80               LDA $30
 90               STA ADDRESS+1
100 LOOP          LDY #0
110               LDA (ADDRESS),Y
120               CMP NAME
130               BNE NEXT
140               INY
150               LDA (ADDRESS),Y
160               CMP NAME+1
170               BNE NEXT
180               BEQ FINISH
190 NEXT          LDY #3
200               LDA (ADDRESS),Y
210               PHA
220               DEY
230               LDA (ADDRESS),Y
240               ADC ADDRESS
250               STA ADDRESS
260               PLA
270               ADC ADDRESS+1
280               STA ADDRESS+1
290               LDA $31
300               CMP ADDRESS
310               BNE LOOP
320               LDA $32
330               CMP ADDRESS+1
340               BNE LOOP
350               LDX #0
360 LOOP2         LDA TEXT,X
370               JSR OUTPUT
380               INX
390               CPX #$11
400               BNE LOOP2
410               BEQ FINISH
420 TEXT          TXT "ARRAY NOT PRESENT"
430 FINISH        RTS
```

Program 6.14. Array search routine.

252 ($FC) from BASIC. The BASIC test routine, Program 6.15, shows how this is done in lines 80 and 90. On calling the machine code routine, the array name codes are searched for and, if found, the address is stored in locations $FB and $FE. The pointer ADDRESS (2 bytes) can then be utilised instead of $2F and $30 in any of the sort routines in Chapter 6. It should be pointed out that the locations used in Program 6.14 will clash, in its present form, with locations used in the previous sort routines. The solution is simple: assign different locations to the labels in lines 20 and 30 and assemble at, say, $C400.

```
10 REM TEST PROGRAM FOR ARRAY SEARCH
20 A%=0:B%=0:C%=0:D%=30
30 REM DIMENSION UP 3 ARRAYS
40 DIM A$(D%)
50 DIM B$(D%)
60 DIM C$(D%)
70 REM FIND START ADDRESS OF ARRAY B$
80 POKE 251,ASC("B")
90 POKE 252,128
100 SYS 49152
110 A%=PEEK(47)+PEEK(48)*256
120 B%=PEEK(253)+PEEK(254)*256
130 C%=PEEK(49)+PEEK(50)*256
140 IF B%=C% THEN 180
150 PRINT"ARRAY SPACE START="A%
160 PRINT"ARRAY ADDRESS     ="B%
170 PRINT"ARRAY SPACE END   ="C%
180 END
```

Program 6.15. BASIC test program for array searching.

Summary

1. Sorting routines, where large numbers of elements are involved, can be painfully slow in BASIC. Machine code versions are much faster.
2. Even the bubble sort is often fast enough if it is in machine code.
3. Understanding program listings in this chapter demands knowledge of the way BASIC stores array variables.
4. BASIC integer arrays are stored in consecutive two-byte form commencing with a seven byte header – 2 bytes for the array name, 2 bytes for the next array pointer, 1 byte for the number of dimensions and the last 2 bytes for array size.

5. BASIC string arrays are stored in ASCII characters. The array header contains details of the length and the address of where the strings are stored and is known as the string information block.
6. The 'merge sort', although more complex and requiring more coding lines, is still based on the bubble sort but capitalises on the advantages offered by a progressive increase in order. It is much faster than the bubble sort where large numbers of elements are involved.
7. The course of a program can often be understood more easily if supported by a 'trace table'.
8. To test out machine code sort routines given in this chapter, BASIC test programs are provided. These pass the parameters entered from the keyboard, generate random arrays and display them again after sorting.
9. BASIC stores floating point numbers in 5 byte form, four for the mantissa (the significant digits) and one for the exponent.
10. Records can use either fixed-length fields and be one long single string element or a two-dimensional string array in row/column format.

Self test

6.1 Name at least five factors which you consider might influence the processing power of a computer.

6.2 Develop a comprehensive merge sort program which will sort signed integers, strings or floating point arrays.

6.3 Develop a comprehensive bubble sort program which will sort signed integers, strings or unsigned floating point numbers.

6.4 Develop a routine which will sort signed floating point numbers.

6.5 Develop a bubble sort routine which will sort signed integers.

6.6 Develop a routine, called from BASIC, which will sort signed integers into either ascending or descending order as requested.

Chapter Seven
High Resolution Graphics

Applications

Apart from the obvious application to computer games, high resolution graphics are invaluable as diagrammatic aids. Graphs, pi charts, bar charts and historgrams can convey information in concise and easily assimilated form. Although the graphics keys on the Commodore 64 are easy to use, the picture resolution is limited by the large 8 × 8 'pixel' size. High resolution graphics, although more difficult to handle than the graphic keys, can produce finer grain displays. Virtually all teaching programs benefit if screen diagrams are used to support textual material and, over the last few years, it has become fashionable to include diagrams and charts in much of the software designed for general business use.

High resolution bit-mapping

High resolution bit-mapped graphics modes are difficult to use in BASIC. Furthermore, the execution speed is painfully slow unless the optional cartridge is fitted with the Extended BASIC repertoire. The relative speed and freedom that assembly language programing offers can help us to utilise these modes more efficiently. By arming ourselves with a few machine code routines it is possible to tailor high resolution graphics screens quite easily.

Standard high resolution bit-mapped mode

This section is based around a high-resolution graphics utility which can light individual pixels specified by simple X,Y screen coordinates. The routine Program 7.1 can be called from BASIC or used within an assembly language program. For those without an assembler the object code is listed in Hex Dump 7.1. This can be loaded directly from $C000 using the hex loader given in Chapter 4 (Program 4.1). The routine is followed by a

detailed breakdown of the coding. An example of its use is also provided in the form of a simple BASIC graph plotter, Program 7.2, which calls the utility from within the program.

In standard high resolution mode, the Commodore 64 sets aside 8K of memory in order to bit-map the entire screen with a resolution of 320 by 200 pixels. This area of memory is called, not surprisingly, the *bit map*. The colours available in this mode are limited to two and are determined by the contents of screen memory. The *upper* nybble supplies the colour information code of any pixel represented by a binary 1 in the bit map. The *lower* nybble supplies the colour code of any pixel represented by a zero in the bit map. The whole process is coordinated by the VIC-II chip and is transparent to the user. The standard high resolution bit-mapped mode is enabled by setting bit 5 of the VIC-II control register to a '1'. The base address of the VIC-II register block is located at $D000 and the control register is located at $D011. The following code will do this:

```
LDA $D011
ORA #$20
STA $D011
```

When the above code is assembled and executed, the screen will be filled with garbage. Unfortunately, there are no routines resident in the Commodore 64 to clear the bit map area or to load up the screen memory with the desired colour codes, so we need to write our own.

Clearing the bit map area, consisting of 8K bytes of memory, would take many seconds using a series of POKEs from BASIC. This delay is clearly unacceptable. A machine code routine can handle this instantaneously – from the user's point of view. Lines 650 to 810 of Program 7.1 make up a subroutine capable of performing this efficiently. Refer to the program breakdown section for details if you are not sure how it works.

A further complication is that screen memory must be set up prior to using any hi-res graphics screens. The reason is that the VIC-II gets its colour information from this area of memory. This is a similar exercise to the above but instead of clearing out memory to zeros the actual colour codes need to be placed in memory. Lines 830 to 1010 of Program 7.1 show how to do this. The location labelled SCRCOL is used to hold the two-colour choice and is best set up in BASIC prior to calling the routine by POKEing location 254. Referring to Program 7.2 (the BASIC graph-plotting example) line 30 shows POKE 254,7. This will set the colours to black on cyan – a good combination of colours for this sort of use. The high nybble is set to zero, so the foreground colour will be black. The low nybble is set to 7, thus specifying cyan for the background colour. If you are using the routine from within an assembly language program the following will need to be added to Program 7.1 at the initialisation stage:

```
LDA #7
STA #$FE
```

Again refer to the program breakdown section for details of the coding.

In order to complete the utility, a subroutine is required which will set the appropriate bit within the bit map according to the specified X,Y coordinates. Figure 7.2 shows the XY coordinate system required. Referring to Fig. 7.1, it is seen that the bit-mapped area of memory is laid out in blocks of eight sequential locations. These correspond to the 25 rows

$0	$8		$138
$1	$9		$139
$2	$A		$13A
$3	$B		$13B
$4	$C		$13C
$5	$D		$13D
$6	$E		$13E
$7	$F		$13F
$140	$148		$278
$141	$149		$279
$142	$14A		$27A
$143	$14B		$27B
$144	$14C		$27C
$145	$14D		$27D
$146	$14E		$27E
$147	$14F		$27F
$3000	$3008		$3138
$3001	$3009		$3139
$3002	$300A		$313A
$3003	$300B		$313B
$3004	$300C		$313C
$3005	$300D		$313D
$3006	$300E		$313E
$3007	$300F		$313F

Fig. 7.1. Bit map.

of 40 programmable characters as seen by the VIC-II chip. This layout is chosen in most microcomputers because it is convenient for text. However, for bit mapping, where we deal in XY coordinates, it is not the ideal organisation of memory.

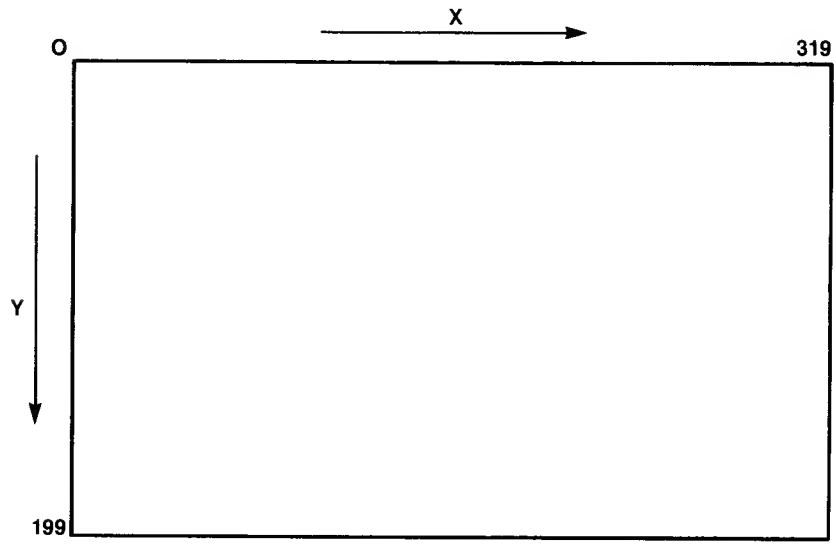


Fig. 7.2. The XY coordinate system.

The difference between equivalent address positions in any adjacent block in the Y direction is $140 or 320 decimal. Similarly, in the X direction, the difference is 8. Therefore the equation to calculate a unique bit map address from an XY coordinate is given by:

ADDRESS = BASE + ROW*320 + LINE +CHAR*8

where
BASE = the bit map start address. (This is usually set to $2000.)
ROW = INT(Y/8), the block of eight or character row number (0 to 24) containing the Y coordinate.
LINE = (Y AND 7). This is the position within the character or block of eight bytes described earlier which contains the Y coordinate (0 to 7).
CHAR = INT(X/8). This is the position of the block or character, within the row, which contains the X coordinate (0 to 39).

Rearranging the equations

The above equation is not suited to machine coding in its present form. We need to rearrange the equation so that all multipliers are, as far as possible, equal to *exact powers of two*. The reason for this is that any multiplication or division simplifies to shifting bits left or right respectively. This saves us coding multiplication or division algorithms which would be relatively slow due to their general purpose nature. The rearrangement can be conveniently achieved as follows:

ADDRESS = BASE +40*(ROW*8) + LINE + CHAR*8
ADDRESS = BASE + 32*(ROW*8) + 8*(ROW*8) + LINE + CHAR*8

Substituting the equations for ROW,LINE, and CHAR we have:

ADDRESS = $2000 + 32*(INT(Y/8)*8) + 8*(INT(Y/8)*8) + (Y AND 7) + 8*INT(X/8)

The above equation may appear unnerving to code initially, but in fact it is relatively easy. All INT(Y/8) entails is shifting Y right 3 times, thus dividing by 8 and losing the remainder (3 least significant bits of Y). Multiplying by 8, giving INT(Y/8)*8, is then achieved by shifting left 3 times. The net result of all this is simply to lose the 3 least significant bits of Y. They have fallen off the end. A simple way of arriving at the same result is to mask out the 3 least significant bits of Y with AND #$F8. This can be accomplished with:

```
LDY YCOORD
AND #$F8
```

Similarly, INT(X/8)*8 can be coded by ANDing the lower byte of X with #$F8.

```
LDA XCOORD
AND #$F8
```

Remember that the specified X coordinate will occupy two bytes in this particular graphics mode (0 to 319).

The 32*(INT(Y/8)*8) expression can be arrived at by shifting the result of INT(Y/8)*8 left 5 times. However, if you can imagine a two-byte result, shifting right 3 times and storing it as the high-byte of the result is the exact equivalent and uses less instructions.

```
LDA #0
STA LOC
LDA YCOORD
LSR A
LSR A
LSR A
STA LOC+1
```

We can do this because the 3 least significant bits of INT(Y/8)*8 have been masked out by the previous operation. The low-byte of the result can be cleared to zero.

Two further shifts right, of the two-byte 32*(INT(Y/8)*8) result, will divide by 4 giving 8*(INT(Y/8)*8). However, there is a complication. We must rotate the carry into the low-byte of the result each time. The LSR and ROR instructions, respectively, are needed for this as shown below:

```
LSR A
ROR LOC
LSR A
ROR LOC
```

The remaining easily coded expression (Y AND 7) completes the equation.

```
LDA YCOORD
AND #7
```

All that is needed now is to add the whole lot together to give the required bit map address. Refer to lines 360 to 630 of Program 7.1 for details of one way to code the above. A detailed breakdown of the utility is also provided.

The above text points the way to calculating the address of the bit map. We now have to set the appropriate bit corresponding to the XY coordinate within the calculated location. This can be achieved by generating a mask and ORing with the contents of the above location. The mask can be constructed by setting the carry and rotating right the required number of times. The loop counter can be initialised, from the 3 least significant bits of X, which is the X co-ordinate value. The following piece of code will do this:

```
        LDA XCOORD
        AND #7
        TAX
        SEC
        LDA #0
SHIFT   ROR A
        DEX
        BPL SHIFT
        STA MASK
```

```
 10 ! HI-RESOLUTION GRAPHICS UTILITY
 20 XCOORD       =  $FB
 30 YCOORD       =  $FD
 40 SCRCOL       =  $FE
 50 BMPAGE       =  $FF
 60 MASK         =  $59
 70 LOC          =  $5A
 80 STORE        =  $5C
 90 SCRPAGE      =  $COBO
100 *=$C000
110 !
120             LDA #$20
130             STA BMPAGE
140             LDA #4
150             STA SCRPAGE
160             LDA $D018
170             ORA #8
180             STA $D018
190             LDA $D011
200             ORA #$20
```

```
210                 STA $D011
220                 JSR CLEARMAP
230                 JSR SCRFILL
240                 RTS
250 !
260 PLOTBIT         LDA XCOORD
270                 AND #7
280                 TAX
290                 SEC
300                 LDA #0
310                 STA LOC
320 SHIFT           ROR A
330                 DEX
340                 BPL SHIFT
350                 STA MASK
360                 LDA XCOORD
370                 AND #$F8
380                 STA STORE
390                 LDA YCOORD
400                 LSR A
410                 LSR A
420                 LSR A
430                 STA LOC+1
440                 LSR A
450                 ROR LOC
460                 LSR A
470                 ROR LOC
480                 ADC LOC+1
490                 STA LOC+1
500                 LDA YCOORD
510                 AND #7
520                 ADC LOC
530                 ADC STORE
540                 STA LOC
550                 LDA LOC+1
560                 ADC XCOORD+1
570                 ADC BMPAGE
580                 STA LOC+1
590                 LDY #0
600                 LDA (LOC),Y
610                 ORA MASK
620                 STA (LOC),Y
630                 RTS
640 !
650 CLEARMAP        LDA BMPAGE
```

```
 660                STA STORE+1
 670                LDA #0
 680                STA STORE
 690                LDX #$1F
 700 LOOP           LDY #0
 710 LOOP2          STA (STORE),Y
 720                DEY
 730                BNE LOOP2
 740                INC STORE+1
 750                DEX
 760                BNE LOOP
 770                LDY #$3F
 780 LOOP3          STA (STORE),Y
 790                DEY
 800                BPL LOOP3
 810                RTS
 820 !
 830 SCRFILL        LDA #0
 840                STA STORE
 850                LDA SCRPAGE
 860                STA STORE+1
 870                LDA SCRCOL
 880                LDX #3
 890 BLOCK          LDY #0
 900 CYCLE          STA (STORE),Y
 910                DEY
 920                BNE CYCLE
 930                INC STORE+1
 940                DEX
 950                BNE BLOCK
 960                STA (STORE),Y
 970                LDY #$E7
 980 NEXT           STA (STORE),Y
 990                DEY
1000                BNE NEXT
1010                RTS
```

Program 7.1. High resolution graphics utility.

```
10 REM TEST PROGRAM FOR HI-RESOLUTION
20 REM GRAPHICS ROUTINES (BIT MAPPING)
30 POKE254,7:REM PASS COLOUR NYBBLES
40 SYS49152:REM CALL INIT ROUTINES
50 FOR X=0 TO 319
60 Y%=100+80*SIN(X/20)
```

```
70 XH%=X/256
80 XL%=X-(XH%*256)
90 REM PASS PARAMETERS
100 POKE251,XL%
110 POKE252,XH%
120 POKE253,Y%
130 SYS49184:REM CALL PLOTBIT
140 NEXT X
150 GOTO150
```

Program 7.2. Test program for high resolution graphics routine.

```
C000 A9 20 85 FF A9 04 8D B0
C008 C0 AD 18 D0 09 08 8D 18
C010 D0 AD 11 D0 09 20 8D 11
C018 D0 20 62 C0 20 B0 C0 60
C020 A5 FB 29 07 AA 38 A9 00
C028 85 5A 6A CA 10 FC 85 59
C030 A5 FB 29 F8 85 5C A5 FD
C038 4A 4A 4A 85 5B 4A 66 5A
C040 4A 66 5A 65 5B 85 5B A5
C048 FD 29 07 65 5A 65 5C 85
C050 5A A5 5B 65 FC 65 FF 85
C058 5B A0 00 B1 5A 05 59 91
C060 5A 60 A5 FF 85 5D A9 00
C068 85 5C A2 1F A0 00 91 5C
C070 88 D0 FB E6 5D CA D0 F4
C078 A0 3F 91 5C 88 10 FB 60
C080 A9 00 85 5C AD B0 C0 85
C088 5D A5 FE A2 03 A0 00 91
C090 5C 88 D0 FB E6 5D CA D0
C098 F4 91 5C A0 E7 91 5C 88
C0A0 D0 FB 60
```

Hex Dump 7.1. Object code for Program 7.

Breakdown of Program 7.1

Lines 10 to 90 assign named locations.

Line 100 causes assembly at $C000.

Lines 120 to 130 set the labelled location BMPAGE to $20 which contains the base address page of the bit map.

Lines 140 to 150 set the labelled location SCRPAGE to 4, which contains the base address page of screen memory.

Lines 160 to 180 place the bit map at $2000.

Lines 190 to 210 enable the bit map mode.

Line 220 calls the subroutine CLEARMAP, which clears the bit map area of memory.
Line 230 calls the subroutine SCRFILL, which sets up screen memory.
Lines 260 to 350 clear the location LOC and produce the mask for setting the XY coordinate bit.
Lines 360 to 580 calculate the XY coordinate address LOC (2 bytes). (See earlier text for details.)
Lines 590 to 620 set the appropriate bit by ORing the mask with the calculated address.
Lines 650 to 680 initialise STORE and STORE+1 to the base address of the bit map as set in the location BMPAGE.
Line 690 sets the X register to #$1F which is the page counter. This is set to the nearest whole number of pages (256 byte blocks) to be cleared in the bit map.
Lines 700 to 760 form a loop which clears memory a page at a time using indirect indexed addressing.
Lines 770 to 800 form a loop which clears the odd $3F bytes of the bit map remaining.
Lines 830 to 860 initialise STORE and STORE+1 to the base address of screen memory as set in SCRPAGE.
Line 870 loads the accumulator with the combination of colours set up in the location SCRCOL. This is POKEd in from BASIC for maximum flexibility in colour choice as explained earlier.
Lines 880 to 1000 load up screen memory in a similar manner to the CLEARMAP subroutine.

Test program

Program 7.2 is a routine which not only tests Program 7.1 but shows how the routine can be called from BASIC.
Line 30 passes the colour nybble codes by POKEing location 254. In this case black on cyan is used.
Line 40, SYS49152 calls the initialisation routine. This sets up the bit map mode, clears the bit map area of memory and sets up screen memory.

Lines 50 to 140 are a loop, responsible for calculating and passing parameters for use by the routine. The program plots a simple sinusoidal function for illustrative purposes. A far more elaborate graph plotting program may be written in BASIC using these machine code routines. Notice that 2 bytes are required for the X coordinate. The values XL% and XH% are POKEd into locations 251 and 252 respectively. The Y coordinate value is POKEd into location 253. A SYS49184 call is needed to plot the appropriate bit.
Line 150 is a loop which locks the program so that scrolling will not occur on completion of graph plotting.

Multicolour bit map mode

This mode has only half the horizontal resolution of the mode previously described. However, the choice of colours available in any 8 by 8 dot programmable character space is increased from 2 to 4. Multicolour bit map mode is enabled by the following:

```
LDA $D018
ORA #8
STA $D018
LDA $D011
ORA #$20
STA $D011
```

and disabled by:

```
LDA $D018
AND #$DF
STA $D018
LDA $D011
AND #$EF
STA $D011
```

This mode utilises 160 by 200 pixels and differs from the standard hi-resolution mode. A bit pair, rather than a solitary bit, determines which pixel is lit. Furthermore, the 2 bit code, to which the bit pair is set, determines the colour of the pixel. As in standard hi-resolution bit-mapped mode, there is a one to one relationship between the 8K of memory used and what is viewed on the screen. The setting of a bit pair to a particular code instructs the VIC-II chip from where it is to get its colour information. The codes are as follows:

Bit pair	Colour information taken from:
00	Background colour location $D021
01	Upper 4 bits of screen memory
10	Lower 4 bits of screen memory
11	Lower 4 bits of colour memory

Each bit pair can specify one of 4 colours as long as the various areas of memory, given above, are each loaded with one of the standard colour codes we are used to in BASIC. These are given on page 139 of the User

Manual supplied with the machine. For example, setting each location of screen memory to $25 will set the upper nybbles to 2 and the lower nybbles to 5, specifying red and green respectively. Thus, if any particular bit pair is set to 01, a red pixel will be lit. Alternatively, a bit pair code 10, will cause a green pixel to light up. Notice how hexadecimal notation simplifies the setting of nybbles. It would be tedious and error-prone to use decimal notation.

The bit map memory needs to be cleared prior to use and screen memory should be set up in a similar manner to that shown above. Additionally, colour memory can be set up to a specific colour code if four-colour mode is required. You may appreciate that this entails access to a large number of separate locations and may well take the time necessary to drink a cup of tea if BASIC were to be used. However, short machine code routines perform this task instantaneously as far as the user is concerned.

Experimental routines

Program 7.3 is a set of routines for experimenting with the multicolour bit mapped mode. The subroutines are capable of plotting individual pixels or lines, specified by XY coordinates and supplied colour codes. The routines are intended to be used from within assembly language programs and cannot be called from BASIC in their present form. However, modifications can easily be made by passing parameters such as X and Y coordinates via the POKE statement.

The subroutine CLEARMAP is identical to that in Program 7.1 and is used to clear the bit map area of memory. MEMFILL is a subroutine used for setting up chosen colour information into screen memory and/or colour memory. The parameters are passed via the X and Y registers. The X register should contain the base page of the area of memory to be set up. The Y register should contain the chosen colour code information that is placed into each location of the intended area of memory.

PLOTPIXEL is a similar subroutine to PLOTBIT described earlier. The essential differences are due to the reduced horizontal resolution of 160 and increased selection of colours. In multicolour bit-mapped mode, the equation that specifies the address of a given XY coordinate is:

$$\text{ADDRESS} = \text{BASE} + \text{ROW}*320 + \text{LINE} + \text{CHAR}*8$$

where
BASE = $2000 (default)
ROW = INT (Y/8) as before
LINE = (Y AND 7) as before
However, CHAR = INT (X/4) this time since only half the horizontal resolution is available. There are still 40 characters a line but the value of X

can only be between 0 and 159. Again, this can be simplified to a form which is more easily coded:

$$\begin{aligned}\text{ADDRESS} &= \text{BASE} + 40*(\text{ROW}*8) + \text{LINE} + \text{CHAR}*8\\ &= \text{BASE} + 32*(\text{ROW}*8) + 8*(\text{ROW}*8) + \text{LINE} + \text{CHAR}*8\end{aligned}$$

Expanding,

$$\text{ADDRESS} = \$2000 + 32*(\text{INT}(Y/8)*8) + 8*(\text{INT}(Y/8)*8) + (Y \text{ AND } 7) + 8*\text{INT}(X/4)$$

The above equation differs from that derived for PLOTBIT only in the last term. The INT(X/4) term can be coded by shifting right twice, thus dividing by 4 and losing the two least significant bits. The term 8*INT(X/4) is obtained by shifting the result left three times. This, in effect, is equivalent to ANDing out the two least significant bits of X and shifting left once. The following code performs this:

```
LDA XCOORD
AND #$FC
ASL A
ROL STORE+1
STA STORE
```

Notice that, although XCOORD is a single byte number, the result of 8*INT(X/4) requires two bytes STORE and STORE+1. This is because the most significant bit of XCOORD is shifted via the carry into the least significant bit of STORE+1. The rest of the coding for the above equation is identical to that described for Program 7.1. The whole lot is then added together as before and the calculated address placed in LOC and LOC+1.

Setting the mask

The mask byte used in setting the appropriate bit pair within the calculated address is, unfortunately, more complex. The essential coding is:

```
        LDA XCOORD
        AND #3
        TAX
        LDA COLCODE
        CLC
LOOP    ROR A
        ROR A
        ROR A
        DEX
        BPL LOOP
        STA MASK
```

XCOORD is loaded into the accumulator and the five most significant

bits are ANDed out. The result in range 0 to 3 is placed in the X register which is acting as a loop counter. The bit pair, specifying where the VIC-II chip gets its colour information from, is previously selected and stored in the location labelled COLCODE. This pattern is loaded into the accumulator and the loop initialised. This is performed by clearing the carry flag and rotating the accumulator one place to the right. The least significant bit of the bit pair is now in the carry and the most significant bit of the bit pair is placed in the least significant bit of the accumulator. Two further rotate right instructions will cause the bit pair to enter the accumulator from the left. Each further cycle of the loop will cause the bit pair to shift right twice. According to the number of cycles of the loop (0 to 3), the bit pair will be set to any one of the four possible mask positions. The number of cycles is dictated by the two least significant bits of XCOORD. The mask can then be ORed with the contents of the calculated address LOC by indirect indexed addressing.

```
 10 ! MULTICOLOUR BIT MAPPING
 20 !PLOTTING INDIVIDUAL PIXELS
 30 XCOORD       =  $FB
 40 YCOORD       =  $FC
 50 COLCODE      =  $FD
 60 SCRMEM       =  $FE
 70 COLMEM       =  $FF
 80 BMPAGE       =  $59
 90 LOC          =  $5A
100 STORE        =  $5C
110 SCRPAGE      =  $5E
120 MASK         =  $50
130 *=$C000
140 !
150              LDA #$20
160              STA BMPAGE
170              LDA #4
180              STA SCRPAGE
190              LDA #$25
200              STA SCRMEM
210              LDA #4
220              STA COLMEM
230              LDA $D018
240              ORA #8
250              STA $D018
260              LDA $D011
270              ORA #$20
280              STA $D011
290              LDA $D016
300              ORA #$10
```

```
310               STA $D016
320               LDA #6
330               STA $D020
340               LDA #7
350               STA $D021
360               JSR CLEARMAP
370               LDX SCRPAGE
380               LDY SCRMEM
390               JSR MEMFILL
400               LDX #$D8
410               LDY COLMEM
420               JSR MEMFILL
430 !
440               LDA #3
450               STA COLCODE
460               LDX #0
470               LDY #100
480               LDA #160
490               JSR HLIN
500               LDA #2
510               STA COLCODE
520               LDX #80
530               LDY #0
540               LDA #200
550               JSR VLIN
560 FIN           JMP FIN
570 !
580 HLIN          STX XCOORD
590               STY YCOORD
600               TAX
610 REP           JSR PLOTPIXEL
620               INC XCOORD
630               DEX
640               BNE REP
650               RTS
660 !
670 VLIN          STX XCOORD
680               STY YCOORD
690               TAX
700 REP2          JSR PLOTPIXEL
710               INC YCOORD
720               DEX
730               BNE REP2
740               RTS
750 !
```

```
 760 PLOTPIXEL   LDA XCOORD
 770             AND #3
 780             STA STORE
 790             LDA #0
 800             STA LOC
 810             STA STORE+1
 820             LDA COLCODE
 830             CLC
 840             ROR A
 850 LOOP4       ROR A
 860             ROR A
 870             DEC STORE
 880             BPL LOOP4
 890             STA MASK
 900             LDA XCOORD
 910             AND #$FC
 920             ASL A
 930             ROL STORE+1
 940             STA STORE
 950             LDA YCOORD
 960             LSR A
 970             LSR A
 980             LSR A
 990             STA LOC+1
1000             LSR A
1010             ROR LOC
1020             LSR A
1030             ROR LOC
1040             CLC
1050             ADC LOC+1
1060             STA LOC+1
1070             LDA YCOORD
1080             AND #7
1090             ADC LOC
1100             ADC STORE
1110             STA LOC
1120             LDA LOC+1
1130             ADC STORE+1
1140             ADC BMPAGE
1150             STA LOC+1
1160             LDY #0
1170             LDA (LOC),Y
1180             ORA MASK
1190             STA (LOC),Y
1200             RTS
```

```
1210 !
1220 CLEARMAP     LDA BMPAGE
1230              STA STORE+1
1240              LDA #0
1250              STA STORE
1260              LDX #$1F
1270 LOOP         LDY #0
1280 LOOP2        STA (STORE),Y
1290              DEY
1300              BNE LOOP2
1310              INC STORE+1
1320              DEX
1330              BNE LOOP
1340              LDY #$3F
1350 LOOP3        STA (STORE),Y
1360              DEY
1370              BPL LOOP3
1380              RTS
1390 !
1400 MEMFILL      LDA #0
1410              STA STORE
1420              STX STORE+1
1430              TYA
1440              LDX #3
1450 BLOCK        LDY #0
1460 CYCLE        STA (STORE),Y
1470              DEY
1480              BNE CYCLE
1490              INC STORE+1
1500              DEX
1510              BNE BLOCK
1520              STA (STORE),Y
1530              LDY #$E7
1540 NEXT         STA (STORE),Y
1550              DEY
1560              BNE NEXT
1570              RTS
```

Program 7.3. Multicolour bit mapping.

Two subroutines are incorporated, capable of drawing vertical or horizontal lines. These are called VLIN and HLIN respectively. Three parameters must be passed in order to draw a line. They are the X and Y coordinates, where the line is to start, and the length of the line to be drawn. The length can be 1 to 200 in the Y dimension or 1 to 160 in the X dimension. Parameters are passed to the subroutine

by means of the X register for the X coordinate and the Y register for the Y coordinate. The length parameter is passed in the accumulator. The parameters should be set up in the various registers *prior* to calling the subroutine. An example of using VLIN is given below:

```
LDA #2
STA COLCODE
LDX #80
LDY #0
LDA #200
JSR VLIN
```

The above code draws a vertical line from top to bottom about halfway across the screen. The selected colour for the line must be set, using the bit pair code (0 to 3), and stored in the location COLCODE. This is required by the PLOTPIXEL subroutine. Note that these are not the colour codes we use in BASIC but the pointer code to where this information may be found (in the table given earlier). The subroutines HLIN and VLIN themselves are just simple loops that repetitively plot pixels by calling the PLOTPIXEL subroutine.

The lines of source code 440 to 560 show how to use the routines by drawing a pair of coloured cross hair-lines. Far more elaborate graphics can be programmed using these routines but, sadly, there is insufficient space for the subject in this book. For instance, Program 7.3 can be used in the same manner as Program 7.1 for plotting hi-res graphs, etc.

Breakdown of Program 7.3

Lines 30 to 120 assign labels to often used locations.
Line 130 instructs assembly starting at location $C000.
Lines 150 to 180 set up the base pages of the bit map and screen memory areas of memory respectively.
Lines 190 to 200 set up the colour nybbles pattern to be stored in each byte of screen memory.
Lines 210 to 220 set up the colour code nybble to be stored in each byte of colour memory.
Lines 230 to 250 place the bit map base address at $2000.
Lines 260 to 310 set up multicolour bit map mode.
Lines 320 to 330 set the border colour to blue.
Lines 340 to 350 set the background colour #0 to yellow.
Line 360 executes the clear bit map subroutine CLEARMAP.
Line 370 to 390 sets up screen memory with the colour code pattern selected in line 190 via the subroutine MEMFILL. Parameters are passed to MEMFILL in the X and Y registers. The X register

contains the base page of memory to be filled and the Y register passes the colour code pattern contained in SCRMEM.
Lines 400 to 420 set up colour memory with the colour code selected in line 210. The X register passes the base page of colour memory and the Y register passes the colour code contained in COLMEM.
Lines 440 to 550 show how to use the routines, as discussed earlier.
Line 560 is a lock-up loop to prevent screen scrolling.
Lines 580 to 650 contain the HLIN subroutine. This stores the parameters passed and calls the PLOTPIXEL subroutine many times within a simple loop to draw a horizontal line of specified length starting at a chosen XY coordinate.
Lines 670 to 740 contain the VLIN subroutine and is similar to the HLIN subroutine above but draws a vertical line.
Lines 760 to 1200 contain the PLOTPIXEL subroutine and is a final polished up version of the coding detailed earlier.
Lines 1220 to 1390 are the CLEARMAP subroutine, identical to that used in Program 7.1.
Lines 1400 to 1570 contain the MEMFILL subroutine which has the dual purpose of filling both screen memory and colour memory with the desired colour codes. This routine is essentially the same as SCRFILL described in the Program 7.1 breakdown except that parameters are passed via the X and Y registers. The base page of memory to be filled is passed in the X register and the filler byte pattern is passed in the Y register.

Drawing hi-resolution static shapes

This section is built around a high resolution shape-drawing routine using the multicolour bit map mode. The routines could also be used for producing software sprites. However, this is unnecessary in the case of the Commodore 64 since hardware sprite handling is provided. The basic idea is to draw a complex multicolour shape at any XY screen coordinate. The routines are designed for drawing any number of different shapes from labelled data tables. The data, describing the shape to be drawn, will be placed at the end of Program 7.4 and can easily be modified to draw any designed shape.

The routine makes use of what should now be familiar methods of translating an XY coordinate to an absolute bit map address. The subroutine responsible is labelled ADDRESS and differs little from that developed in the previous listing. The subroutines MEMFILL and CLEARMAP are used to clear and prepare the viewed screen in the manner discussed previously.

In multicolour bit map mode, each byte of bit map corresponds to 4 consecutive horizontal pixels as seen on the screen. In order to cut down on the number of data elements required to specify a shape, a byte is considered as the fundamental unit. Thus each byte can contain data specifying up to 4 horizontally consecutive pixels of different colours. This is a help, since a lot of information can be packed into the data tables. They can be accessed up to a limit of 255 bytes. Each data byte is constructed, using familiar bit pairs, which point to where the colour information is stored for a particular pixel. The actual colour codes must be previously set up in screen and colour memory as shown in the last section. The codes are repeated below for convenience:

Bit pair code	Colour information taken from:
00	Background colour (location $D021)
01	Upper nybble of Screen memory
10	Lower nybble of Screen memory
11	Lower nybble of colour memory

	COLUMN 0	COLUMN 1	COLUMN 2	COLUMN 3
ROW 0	0	1	2	3
ROW 1	4	5	6	7
ROW 2	8	9	10	11
	12	13	14	15
	16	17		
			50	51
ROW 13	52	53	54	55
ROW 14	56	57	58	59
ROW 15	60	61	62	63

Fig. 7.3. Order in which data bytes are placed in memory.

The subroutine DRAW places the data bytes into the bit map area of memory and hence onto the screen in sequential columns in row by row fashion. Figure 7.3 should clarify this point.

The shape rectangle

It is important here to think of the shape you are designing as being contained in a rectangle. This should present no problem since the background #0 colour can be specified where no shape detail is to be seen. The first two entries of each data table are the *height* and *width* data for the rectangular graphics cell. This data is read in and stored, in the locations HEIGHT and WIDTH respectively, by the DRAW subroutine prior to dumping the rest of the data into the bit map. For instance, in the example data table SHAPE1 shown in program 7.4 the graphics cell is 25 bytes high (25 pixels) by 5 bytes wide (20 pixels). The actual shape is best planned out on grid paper, remembering that each byte in the horizontal direction contains the data of four pixels.

XY coordinates

For ease of use within a larger program, the X and Y coordinate parameters are passed to DRAW using the X and Y registers respectively. The relevant data table, containing the information to be placed into memory, must have its start address placed into the location POINTER and POINTER+1 (low-byte, high-byte) prior to calling DRAW.

The top left-hand corner of the rectangle containing the shape is the reference starting point. If the shape is to be placed at X=20 and Y=100, and the relevant data table base address is labelled SHAPE1, the following code would be required before DRAW is called:

```
LDX #20
LDY #100
LDA #<SHAPE1
STA POINTER
LDA #>SHAPE1
STA POINTER+1
JSR DRAW
```

The X register is loaded with the X coordinate. The Y register is loaded with the Y coordinate. The shape table base address, labelled SHAPE1, is stored in POINTER. Since two bytes are required, the shape table base address must be split into low-byte and high-byte. The MIKRO Assembler uses the format shown above for this, using < to denote the low-byte of the address and > for the high-byte. Other assemblers might have a different

convention, so check this in the instruction manual supplied with the assembler. A call to the subroutine DRAW will then place the shape on the screen at the specified position.

The objective of Program 7.4

Program 7.4 draws two multicoloured shapes on the screen. The first derives its data from the table SHAPE1 and the second from SHAPE2. More can be added using the same techniques and for a number of purposes such as backcloth detail for games, etc, or computer artwork. Even software sprites can be formed with a few minor modifications such as placing a one-pixel border of background #0 round the shape and successively incrementing or decrementing the X and/or Y coordinates.

```
 10 ! MULTICOLOUR BIT MAPPING
 20 !DRAWING MULTIPLE STATIC SHAPES
 30 XCOORD          =  $FB
 40 YCOORD          =  $FC
 50 SCRPAGE         =  $FD
 60 SCRMEM          =  $FE
 70 COLMEM          =  $FF
 80 BMPAGE          =  $26
 90 HEIGHT          =  $57
100 WIDTH           =  $58
110 WCOUNT          =  $59
120 LOC             =  $5A
130 STORE           =  $5C
140 YREG            =  $5E
150 POINTER         =  $4E
160 *=$C000
170 !
180                 LDA #$20
190                 STA BMPAGE
200                 LDA #4
210                 STA SCRPAGE
220                 LDA #$03
230                 STA SCRMEM
240                 LDA #5
250                 STA COLMEM
260                 LDA $D018
270                 ORA #8
280                 STA $D018
290                 LDA $D011
300                 ORA #$20
310                 STA $D011
320                 LDA $D016
```

```
330                 ORA #$10
340                 STA $D016
350                 LDA #4
360                 STA $D020
370                 LDA #7
380                 STA $D021
390                 JSR CLEARMAP
400                 LDX SCRPAGE
410                 LDY SCRMEM
420                 JSR MEMFILL
430                 LDX #$D8
440                 LDY COLMEM
450                 JSR MEMFILL
460 !
470                 LDX #20
480                 LDY #100
490                 LDA #<SHAPE1
500                 STA POINTER
510                 LDA #>SHAPE1
520                 STA POINTER+1
530                 JSR DRAW
540                 LDX #10
550                 LDY #50
560                 LDA #<SHAPE2
570                 STA POINTER
580                 LDA #>SHAPE2
590                 STA POINTER+1
600                 JSR DRAW
610 HALT            JMP HALT
620 !
630 DRAW            STX XCOORD
640                 STY YCOORD
650                 LDY #0
660                 LDA (POINTER),Y
670                 STA HEIGHT
680                 INY
690                 LDA (POINTER),Y
700                 STA WIDTH
710                 LDX #2
720 NEWROW          LDA #0
730                 STA YREG
740                 LDA WIDTH
750                 STA WCOUNT
760                 JSR ADDRESS
770 NEWCOLUMN       TXA
```

```
 780                TAY
 790                LDA (POINTER),Y
 800                LDY YREG
 810                STA (LOC),Y
 820                TYA
 830                CLC
 840                ADC #8
 850                STA YREG
 860                INX
 870                DEC WCOUNT
 880                BNE NEWCOLUMN
 890                INC YCOORD
 900                DEC HEIGHT
 910                BNE NEWROW
 920                RTS
 930 !
 940 ADDRESS        LDA #0
 950                STA LOC
 960                STA STORE+1
 970                LDA XCOORD
 980                ASL A
 990                ASL A
1000                ASL A
1010                ROL STORE+1
1020                STA STORE
1030                LDA YCOORD
1040                LSR A
1050                LSR A
1060                LSR A
1070                STA LOC+1
1080                LSR A
1090                ROR LOC
1100                LSR A
1110                ROR LOC
1120                ADC LOC+1
1130                STA LOC+1
1140                LDA YCOORD
1150                AND #7
1160                ADC LOC
1170                ADC STORE
1180                STA LOC
1190                LDA LOC+1
1200                ADC STORE+1
1210                ADC BMPAGE
1220                STA LOC+1
```

```
1230                RTS
1240 !
1250 CLEARMAP       LDA BMPAGE
1260                STA STORE+1
1270                LDA #0
1280                STA STORE
1290                LDX #$1F
1300 LOOP           LDY #0
1310 LOOP2          STA (STORE),Y
1320                DEY
1330                BNE LOOP2
1340                INC STORE+1
1350                DEX
1360                BNE LOOP
1370                LDY #$3F
1380 LOOP3          STA (STORE),Y
1390                DEY
1400                BPL LOOP3
1410                RTS
1420 !
1430 MEMFILL        LDA #0
1440                STA STORE
1450                STX STORE+1
1460                TYA
1470                LDX #3
1480 BLOCK          LDY #0
1490 CYCLE          STA (STORE),Y
1500                DEY
1510                BNE CYCLE
1520                INC STORE+1
1530                DEX
1540                BNE BLOCK
1550                STA (STORE),Y
1560                LDY #$E7
1570 NEXT           STA (STORE),Y
1580                DEY
1590                BNE NEXT
1600                RTS
1610 !
1620 SHAPE1         BYT 25,5
1630                BYT $55,$55,$55,$55,$55
1640                BYT $66,$66,$66,$66,$66
1650                BYT $FF,$FF,$FF,$FF,$FF
1660                BYT $FF,$FF,$00,$FF,$FF
1670                BYT $FF,$FF,$FF,$FF,$FF
```

```
1680                    BYT $66,$66,$66,$66,$66
1690                    BYT $55,$55,$55,$55,$55
1700                    BYT $FF,$FF,$FF,$FF,$FF
1710                    BYT $55,$FF,$FF,$FF,$55
1720                    BYT $66,$FF,$FF,$FF,$66
1730                    BYT $FF,$DB,$DB,$DB,$FF
1740                    BYT $55,$DB,$DB,$DB,$55
1750                    BYT $66,$00,$DB,$00,$66
1760                    BYT $FF,$DB,$DB,$DB,$FF
1770                    BYT $55,$DB,$DB,$DB,$55
1780                    BYT $66,$FF,$FF,$FF,$66
1790                    BYT $FF,$FF,$FF,$FF,$FF
1800                    BYT $FF,$FF,$FF,$FF,$FF
1810                    BYT $55,$55,$55,$55,$55
1820                    BYT $66,$66,$66,$66,$66
1830                    BYT $FF,$FF,$FF,$FF,$FF
1840                    BYT $FF,$FF,$00,$FF,$FF
1850                    BYT $FF,$FF,$FF,$FF,$FF
1860                    BYT $66,$66,$66,$66,$66
1870                    BYT $55,$55,$55,$55,$55
1880 !
1890 SHAPE2             BYT 7,4
1900                    BYT $FF,$FF,$FF,$FF
1910                    BYT $FF,$1B,$1B,$FF
1920                    BYT $FF,$1B,$1B,$FF
1930                    BYT $FF,$FF,$FF,$FF
1940                    BYT $FF,$1B,$1B,$FF
1950                    BYT $FF,$1B,$1B,$FF
1960                    BYT $FF,$FF,$FF,$FF
```

Program 7.4. Drawing multiple static shapes.

Breakdown of Program 7.4

Lines 30 to 150 assign labels to commonly used locations.

Line 160 defines the assembly beginning (location $C000).

Lines 180 to 210 set up the base pages for the bit map and screen memory.

Lines 220 to 230 set the colour codes for the upper and lower nybbles of screen memory into the location SCRMEM. In this case black is specified for the upper nybble and cyan for the lower nybble.

Lines 240 to 250 set the colour code (green in this case) to be placed in colour memory.

Lines 260 to 340 prepare and set up multicolour bit map mode in the same manner as the previous Program 7.3.

Lines 350 to 360 set the screen border colour to purple.

Lines 370 to 380 set the background #0 or screen background colour to yellow.
Lines 390 to 450 clear the bit map area of memory then fill screen and colour memory with the contents of SCRMEM and COLMEM respectively.
Lines 470 to 600 store the shape table start address pointer, set up the XY coordinate parameters and call the DRAW subroutine. This is repeated so two different shapes can be drawn, obtaining their data from tables SHAPE1 and SHAPE2.
Line 610 loops up the program to prevent screen scrolling.
Lines 630 to 640 store the XY coordinate parameters passed via the X and Y registers in XCOORD and YCOORD.
Lines 650 to 700 pick up the height and width data of the graphics cell and store them in HEIGHT and WIDTH respectively.
Line 710 stores the index to the next data item in the X register. That is to say, the current Y register content is temporarily saved in the X register. The code TYA, followed by TAX, would have done here equally well.
Lines 720 to 730 clear the location YREG at the start of each new graphics cell column. This can be thought of as the column index.
Lines 740 to 750 reset the graphics cell width counter (WCOUNT) with the contents of WIDTH each time a new row is started.
Line 760 calls the subroutine ADDRESS which calculates a bit map address from an XY coordinate.
Lines 770 to 780 transfer the data index, currently stored in the X register, to the Y register, thus enabling the use of indirect indexed addressing.
Line 790 loads the next data item into the accumulator.
Line 800 loads the Y register with the column index stored in YREG.
Line 810 stores the data present in the accumulator in the correct place in the bit map by the use of indirect indexed addressing.
Lines 820 to 850 add the number of bytes offset between adjacent addresses in the X direction to the Y register contents. The result is stored in the column index labelled YREG.
Line 860 increments the data index.
Lines 870 to 880 decrement the width counter and cause a branch to NEWCOLUMN if row is not complete.
Lines 890 to 910 increment the Y coordinate if the row is completed and cause a branch back to NEWROW if the total number of rows are not completed.
Lines 940 to 1230 are the ADDRESS subroutine. This is much the same as similar subroutines described earlier so no further explanation should be necessary.
Lines 1250 to 1410 list the familiar CLEARMAP subroutine.
Lines 1430 to 1600 are the MEMFILL subroutine described in the previous section.
Line 1620 contains the height and width data of SHAPE1.

Lines 1630 to 1870 are the SHAPE1 data table.
Line 1890 contains the height and width data of SHAPE 2.
Lines 1900 to 1960 are the SHAPE2 data table.

Formation and control of sprites

It is assumed, in this section, that readers are already familiar with the formation and control of sprites using BASIC so it is sufficient here to concentrate on assembly language equivalents. The programming of sprites is particularly easy on the Commodore 64 since control simplifies to a series of loops containing POKE statements in BASIC, the machine code equivalent being a succession of LDAs and STAs. However, the task can be tedious and a lot of development time is needed if a full feature-zapping type of game is to be devised. A few examples are given on the formation and control of sprites using assembly language.

There are two sprite modes. They can be formed in either standard resolution two-colour mode (24 by 21 pixels) or in multicolour mode with a reduced horizontal resolution of 12 by 21 pixels.

Standard resolution sprites

A sprite can be designed using decimal methods as described in the User Manual. In assembly language it is far easier to use hexadecimal notation. This is because the data byte is constructed from upper and lower byte nybbles. That is, groups of four bits where each group can have a maximum value of $F or 15 decimal. To translate a relatively complex bit pattern such as 10110111 into decimal would involve calculating 128+32+16+4+2+1= 183. By splitting the bit pattern into groups of four the hex digits $D7 emerge naturally. Those of us who don't shine at mental arithmetic should find hexadecimal a boon. (Refer to Appendix A for practical examples of binary/hex conversions.)

VIC-II addressing

It is always best to assign the VIC-II register base address to a label at the start of a program. For example, BASE=$D000 assigns the base address to the label BASE. With most assemblers, the use of BASE+X, where X is any offset, will now address any of the registers. This method is incorporated in all the examples that follow.

Directly after screen memory there are 8 locations by default starting at $7F8 or 2040 decimal. These locations must contain the block numbers of where the VIC-II chip can find the corresponding sprite data 0 to 7. The block number is ADDRESS/64, so if your sprite #0 data is stored at $340 or 832 decimal the relevant block number is 832/64=13. This is the number that must be loaded into location $7F8.

Address BASE+21 is the sprite enable register. This register is 8 bits wide

and corresponds, lsb to msb, to the sprites 0 to 7. Thus, to turn any sprite on, the relevant bit must be set to '1'. For example, to turn on sprite #0 and sprite #1, the sprite enable register must be set to 3. When visible on the screen, sprites have a priority system: a low numbered sprite will always appear to pass in front of a higher numbered sprite.

What is needed now is to read in the sprite data from a table and load it into the area of memory reserved above. The sprite can then be moved on the screen by altering the contents of locations BASE and BASE+1. These are the current X and Y coordinate registers respectively. Program 7.5 shows the fundamental sprite control arrangement. By studying this and the program breakdown it should be clear how to create and move a single sprite around.

```
 10 !FUNDAMENTAL SPRITE CONTROL
 20 BASE          =  $D000
 30 MEM           =  $FB
 40 *=$C000
 50               LDA #147
 60               JSR $FFD2
 70               LDA #$40
 80               STA MEM
 90               LDA #3
100               STA MEM+1
110               LDA #1
120               STA BASE+21
130               LDA #13
140               STA $7F8
150               LDY #62
160 LOOP          LDA SHAPE,Y
170               STA (MEM),Y
180               DEY
190               BPL LOOP
200 REPEAT        LDX #0
210 LOOP2         LDA BASE+18
220               BNE LOOP2
230               STX BASE
240               STX BASE+1
250               INX
260               CPX #200
270               BNE LOOP2
280               BEQ REPEAT
290 SHAPE         BYT $00,$18,$00
300               BYT $00,$18,$00
310               BYT $00,$7E,$00
320               BYT $00,$FF,$00
```

```
330             BYT $01,$FF,$80
340             BYT $03,$FF,$C0
350             BYT $07,$FF,$E0
360             BYT $0E,$3C,$70
370             BYT $1E,$3C,$78
380             BYT $3E,$3C,$7C
390             BYT $7F,$FF,$FE
400             BYT $FF,$FF,$FF
410             BYT $FF,$FF,$FF
420             BYT $FF,$FF,$FF
430             BYT $78,$00,$1E
440             BYT $31,$00,$1C
450             BYT $1F,$FF,$F8
460             BYT $0F,$F7,$F0
470             BYT $07,$E3,$E0
480             BYT $03,$C1,$C0
490             BYT $01,$81,$80
```

Program 7.5. Fundamental sprite control.

Breakdown of Program 7.5

Lines 20 to 30 assign labels to the commonly used locations.

Line 40 causes assembly to begin at $C000.

Lines 50 to 60 clear the screen. This is the equivalent of PRINT CHR$(147).

Lines 70 to 100 set up the locations MEM and MEM+1 (low-byte, high-byte) to the address where the sprite data is to be stored. This address has been arbitrarily chosen as the cassette buffer ($340) and can hold sufficient data for up to 3 sprites.

Lines 110 to 120 set the sprite enable register to 1, thus turning on sprite #0.

Lines 130 to 140 load the first location, following the default screen memory, with 13. This informs the VIC-II chip that sprite data is located at $7F8 or 2040 decimal.

Line 150 loads the Y register with the number of items in the data table minus 1.

Lines 160 to 190 constitute a loop which dumps the data table contents into the reserved sprite data memory. Indexed addressing with the Y register is used to pick up the data element from the data table. The data is then stored using indirect indexed addressing. the loop is down counting so the last item of data is loaded first then the second to last, etc.

Line 200 initialises the X register to zero.

Lines 210 to 220 form a delay loop which repeatedly reads the faster register $D012 and exits when zero is returned. This reduces flicker since the screen is updated while not in the visible part of the scan.

Lines 230 to 270 simply update the X and Y coordinates of where the sprite

is placed on the screen. LOOP2 controls the sprite across the screen diagonally.
Line 280 causes a branch back to REPEAT thus repeating the control loop LOOP2.
Lines 290 to 490 contain the sprite data table.

More elaborate control routines can be devised but the essential techniques would be clouded by too much uninformative detail.

Separate control of sprites

Program 7.6 shows how to set up two sprites and control them separately. The sprite enable register is set to turn on sprite #0 and sprite #1, therefore 3 is loaded. The sprite data is not placed in the cassette buffer in this example. Instead, the data is placed sequentially in 64 byte blocks from $3000. This is just an alternative area of memory that is not so restrictive in size as the cassette buffer. The block numbers indicating where the sprite data is stored are thus given by 192 and 193 for sprites #0 and #1 respectively. These block numbers are stored directly after screen memory in locations $7F8 and $7F9.

Two or more different data tables can be accessed in the same program by using the method described in the last section on drawing multiple hi-res shapes from tables. The common subroutine SPRITE, used in Program 7.6, can dump each sprite data table into the previously reserved block of memory by simply changing the indirect addresses POINTER and MEM.

```
 10 !SETTING UP TWO SPRITES
 20 BASE          =  $D000
 30 MEM           =  $FB
 40 POINTER       =  $FD
 50 *=$C000
 60               LDA #147
 70               JSR $FFD2
 80               LDA #3
 90               STA BASE+21
100               LDA #192
110               STA $7F8
120               LDA #193
130               STA $7F9
140               LDA #$00
150               STA MEM
160               LDA #$30
170               STA MEM+1
180               LDA #7
190               STA BASE+39
200               LDA #<SHAPE1
```

```
210                 STA POINTER
220                 LDA #>SHAPE1
230                 STA POINTER+1
240                 JSR SPRITE
250                 LDA #$40
260                 STA MEM
270                 LDA #4
280                 STA BASE+40
290                 LDA #<SHAPE2
300                 STA POINTER
310                 LDA #>SHAPE2
320                 STA POINTER+1
330                 JSR SPRITE
340 !
350 REPEAT          LDA #0
360                 STA BASE
370                 STA BASE+1
380                 STA BASE+3
390                 LDA #100
400                 STA BASE+2
410                 LDX #255
420 LOOP2           LDA BASE+18
430                 BNE LOOP2
440                 INC BASE
450                 INC BASE+1
460                 INC BASE+3
470                 DEX
480                 BNE LOOP2
490                 BEQ REPEAT
500 !
510 SPRITE          LDY #62
520 LOOP            LDA (POINTER),Y
530                 STA (MEM),Y
540                 DEY
550                 BPL LOOP
560                 RTS
570 !
580 SHAPE1          BYT $00,$18,$00
590                 BYT $00,$18,$00
600                 BYT $00,$7E,$00
610                 BYT $00,$FF,$00
620                 BYT $01,$FF,$80
630                 BYT $03,$FF,$C0
640                 BYT $07,$FF,$E0
650                 BYT $0E,$3C,$70
```

```
660                BYT $1E,$3C,$78
670                BYT $3E,$3C,$7C
680                BYT $7F,$FF,$FE
690                BYT $FF,$FF,$FF
700                BYT $FF,$FF,$FF
710                BYT $FF,$FF,$FF
720                BYT $78,$00,$1E
730                BYT $31,$00,$1C
740                BYT $1F,$FF,$F8
750                BYT $0F,$F7,$F0
760                BYT $07,$E3,$E0
770                BYT $03,$C1,$C0
780                BYT $01,$81,$80
790 !
800 SHAPE2         BYT $FF,$FF,$FF
810                BYT $FF,$FF,$FF
820                BYT $FF,$00,$FF
830                BYT $FF,$00,$FF
840                BYT $FF,$00,$FF
850                BYT $FF,$00,$FF
860                BYT $FF,$FF,$FF
870                BYT $FF,$FF,$FF
880                BYT 0,0,0,0,0,0,0,0,0,0
890                BYT 0,0,0,0,0,0,0,0,0,0
900                BYT 0,0,0,0,0,0,0,0,0,0
910                BYT 0,0,0,0,0,0,0,0,0,0,0
```

Program 7.6. Setting up two sprites.

Breakdown of Program 7.6

Lines 20 to 40 assign labels to commonly used locations.
Line 50 causes assembly to start at location $C000.
Lines 60 to 70 clear the screen.
Lines 80 to 90 enable sprites #0 and sprite #1.
Lines 100 to 130 store the block numbers where the VIC-II chip can find the data of each sprite. The sprite #0 data is stored in block 192 of memory and the sprite #1 data is stored in block 193. These block numbers are stored at locations $7F8 and $7F9 following screen memory.
Lines 140 to 170 set up the base address pointers MEM (2 bytes) to where the sprite #0 data will be placed.
Lines 180 to 190 set the colour of sprite #0 to yellow (colour code 7).
Lines 200 to 240 set up POINTER (2 bytes) to the base address of the sprite #0 data table labelled SHAPE1. This is followed by a call to the subroutine SPRITE which dumps the data into memory.
Lines 250 to 260 add $40 or 64 decimal to the previous contents of MEM in order to set it to where sprite #1 data will be placed.

Lines 270 to 280 set the colour of sprite #1 to purple (colour code 4).
Lines 290 to 330 set up POINTER (2 bytes) to the base address of the sprite #1 data table labelled as SHAPE2. This is followed by a call to subroutine SPRITE which dumps the data table into memory.
Lines 350 to 400 initialise the X and Y coordinate registers of the VIC-II chip for each sprite.
Lines 410 to 490 form a loop, separately controlling the paths taken by the two sprites. It does this by incrementing a combination of the above registers to provide a control routine for illustrative purposes.
Lines 510 to 550 are the SPRITE subroutine. This picks up data from the data table and stores it in the correct part of memory. Which table, and which part of memory, must be previously set up in POINTER and MEM. The subroutine employs indirect indexed addressing.
Lines 580 to 910 are a pair of data tables corresponding to sprite #0 and sprite #1. Notice, in lines 880 to 910, that unused locations of the sprite data table must be set to zero in order to prevent unpredictable results.

Multicolour sprites

In *Multicolour mode* a sprite is limited to a maximum of four colours. Each bit pair instructs the VIC-II chip on where the actual colour code is stored. This information can be in any one of the registers laid out below and it is the programmer's responsibility to set up the colour codes in them.

Bit pair	Where the colour code is taken from
00	Current screen colour
01	Sprite multicolour register #0 ($D025)
10	Sprite colour register
11	Sprite multicolour register #1 ($D026)

Program 7.7 shows how a multicolour sprite can be added to Program 7.6. The breakdown of Program 7.7 sets out only the details of the additions to Program 7.6.

```
 10 !ADDITION OF A MULTICOLOUR SPRITE
 20 BASE          =  $D000
 30 MEM           =  $FB
 40 POINTER       =  $FD
 50 *=$C000
 60               LDA #147
 70               JSR $FFD2
 80               LDA #7
 90               STA BASE+21
100               LDA #192
```

```
110                 STA $7F8
120                 LDA #193
130                 STA $7F9
140                 LDA #194
150                 STA $7FA
160                 LDA #$00
170                 STA MEM
180                 LDA #$30
190                 STA MEM+1
200                 LDA #7
210                 STA BASE+39
220                 LDA #<SHAPE1
230                 STA POINTER
240                 LDA #>SHAPE1
250                 STA POINTER+1
260                 JSR SPRITE
270                 LDA #$40
280                 STA MEM
290                 LDA #4
300                 STA BASE+40
310                 LDA #<SHAPE2
320                 STA POINTER
330                 LDA #>SHAPE2
340                 STA POINTER+1
350                 JSR SPRITE
360                 LDA #$80
370                 STA MEM
380                 LDA #4
390                 STA BASE+28
400                 LDA #0
410                 STA BASE+41
420                 LDA #4
430                 STA BASE+37
440                 LDA #5
450                 STA BASE+38
460                 LDA #<SHAPE3
470                 STA POINTER
480                 LDA #>SHAPE3
490                 STA POINTER+1
500                 JSR SPRITE
510 !
520 REPEAT          LDA #0
530                 STA BASE
540                 STA BASE+1
550                 STA BASE+3
```

```
 560                STA BASE+5
 570                LDA #100
 580                STA BASE+2
 590                STA BASE+4
 600                STA BASE+5
 610                LDX #255
 620 LOOP2          LDA BASE+18
 630                BNE LOOP2
 640                INC BASE
 650                INC BASE+1
 660                INC BASE+3
 680                INC BASE+4
 690                DEX
 700                BNE LOOP2
 710                BEQ REPEAT
 720 !
 730 SPRITE         LDY #62
 740 LOOP           LDA (POINTER),Y
 750                STA (MEM),Y
 760                DEY
 770                BPL LOOP
 780                RTS
 790 !
 800 SHAPE1         BYT $00,$18,$00
 810                BYT $00,$18,$00
 820                BYT $00,$7E,$00
 830                BYT $00,$FF,$00
 840                BYT $01,$FF,$80
 850                BYT $03,$FF,$C0
 860                BYT $07,$FF,$E0
 870                BYT $0E,$3C,$70
 880                BYT $1E,$3C,$78
 890                BYT $3E,$3C,$7C
 900                BYT $7F,$FF,$FE
 910                BYT $FF,$FF,$FF
 920                BYT $FF,$FF,$FF
 930                BYT $FF,$FF,$FF
 940                BYT $78,$00,$1E
 950                BYT $31,$00,$1C
 960                BYT $1F,$FF,$F8
 970                BYT $0F,$F7,$F0
 980                BYT $07,$E3,$E0
 990                BYT $03,$C1,$C0
1000                BYT $01,$81,$80
1010 !
```

```
1020 SHAPE2       BYT $FF,$FF,$FF
1030              BYT $FF,$FF,$FF
1040              BYT $FF,$00,$FF
1050              BYT $FF,$00,$FF
1060              BYT $FF,$00,$FF
1070              BYT $FF,$00,$FF
1080              BYT $FF,$FF,$FF
1090              BYT $FF,$FF,$FF
1100              BYT 0,0,0,0,0,0,0,0,0,0
1110              BYT 0,0,0,0,0,0,0,0,0,0
1120              BYT 0,0,0,0,0,0,0,0,0,0
1130              BYT 0,0,0,0,0,0,0,0,0,0,0,
1140 !
1150 SHAPE3       BYT $55,$55,$55
1160              BYT $AA,$AA,$AA
1170              BYT $FF,$FF,$FF
1180              BYT $55,$55,$55
1190              BYT $AA,$AA,$AA
1200              BYT $FF,$FF,$FF
1210              BYT $55,$55,$55
1220              BYT $AA,$AA,$AA
1230              BYT $FF,$FF,$FF
1240              BYT 0,0,0,0,0,0,0,0,0,0
1250              BYT 0,0,0,0,0,0,0,0,0,0
1260              BYT 0,0,0,0,0,0,0,0,0,0
1270              BYT 0,0,0,0,0,0,0,0,0
```

Program 7.7. Addition of multicolour sprite.

Breakdown of Program 7.7.

Lines 150 to 160 store the block number (194) where the VIC-II chip can find the additional sprite #2 data. This is placed in the third location following screen memory ($7FA).

Lines 360 to 370 add $40 or 64 decimal to the previous contents of MEM (2 bytes). This sets MEM to the base address of where the sprite #2 data is to be placed.

Lines 380 to 390 sets sprite #2 to multicolour mode by setting bit 2 of the multicolour sprite select register ($D01C).

Lines 400 to 410 set sprite #2 colour to black.

Lines 420 to 430 set sprite multicolour #0 to purple.

Lines 440 to 450 set sprite multicolour #1 to green.

Lines 460 to 500 set up POINTER (2 bytes) to the base address of the sprite #2 data table labelled SHAPE3. This is followed by a call to the SPRITE subroutine which places the data into the reserved memory area.

Lines 600 and 680 increment the sprite X and Y coordinate registers within the simple control loop.

Lines 1150 to 1270 contain the sprite #2 data table.

The routines supplied with this section are the basic methods which enable sprites to be used in assembly language programs. All the other refinements required in order to produce games – such as sprite expanding, collision detection and background priorities – can be easily incorporated by altering the contents of various registers within the VIC-II chip.

Summary

1. The standard high resolution mode uses 8K of memory, called the *bit map* to address 320 × 200 pixels with two colours.
2. The upper nybble of a screen memory byte specifies the colour information code of pixels represented by binary I and the lower nybble, binary 0.
3. The standard high resolution mode is enabled by a 1 in bit 5 position of the VIC-II control register, addressed at $D011.
4. The difference between adjacent block addresses is $140 in the Y direction and 8 in the X direction.
5. The multicolour bit map mode offers a choice of 4 colours but the horizontal resolution is halved to 160 × 200 pixels.
6. When planning shapes in the high resolution mode, it is convenient to consider them enclosed within a rectangle.
7. There are two sprite modes, either two-colour with 24 × 21 or four-colour with 12 × 21 resolution.

Self test

7.1 How many pixels are available in the standard high resolution bit map?

7.2 The standard high resolution bit map uses 8 kilobytes of RAM. How many bits per pixel?

7.3 How do you set the standard high-resolution bit mapped mode?

7.4 In standard high resolution mode, the screen cannot be cleared by zeroing screen locations. Why?

Chapter Eight
TTL Logic and I/O Techniques

Introduction

The Commodore 64 is reasonably well equipped for communication with the outside world. Most users are content to take the various sockets at the side and back for granted. That is to say, devices such as printers, floppy disks, joysticks, modems, etc. are bought complete with ready-made interface leads and simply plugged in. Even the software to drive the devices is often supplied. The users' attitude – which is quite understandable – is often: 'If the device works, why worry about how it works?'. It would be wrong to pretend that do-it-yourself methods, when applied to input/output interfacing, are simple to understand and you would be well advised to take the same attitude unless you already have, or intend to acquire, a *little* knowledge of electronics in general and TTL logic in particular. Fortunately, the study of logic systems does not require a heavy background knowledge of conventional electronic theory. There is no need to wade through masses of alternating current and semiconductor theory.

There are two main reasons why much of this traditional knowledge can be skipped:

(a) Logic is concerned with only two electrical voltages (two-state systems). Traditional electronics, on the other hand, is essentially analogue in nature and so applies to smoothly varying voltages over an enormous range. Consequently, traditional electronics is drenched with mathematics.
(b) Nearly all logic systems, even the most complex, can be constructed on the Meccano principle using a selection of ready-made chips. Interfacing problems between chips of the same family are virtually non-existent because they are purpose-designed to be joined up directly in nose to tail fashion.

There are two families of logic which maintain a kind of friendly rivalry. One is called CMOS (which stands for Complementary Metal Oxide Semiconductor) and the other is called TTL (which stands for Transistor Transistor Logic). These names may seem a little off-putting when met for the first time but they refer only to the kind of transistors used within the

chip. Since chips operate within their own microscopic world, it matters very little to the person using them how the internal semiconductor magic behaves. In general, it is sufficient to know that CMOS chips take very low currents but are not quite as fast or available in such variety as TTL chips. Also, TTL is marginally easier to handle and tends to be more popular in microcomputer work than CMOS. In view of this, only TTL logic will be discussed in this chapter.

Introducing Logic

Simple switching

We start with the assumption that you already possess a smattering of normal electrical knowledge. That is to say, you know the difference between positive and negative, the difference between components in series and parallel and know how to wire up simple circuits consisting of a lamp, battery and one or more switches. Armed with a 4.5 volt dry battery as the power source, you can soon get the feel of simple logic by hooking up the circuits as shown in Fig. 8.1.

The AND function

Figure 8.1(a) shows two switches, marked A and B, in series. Clearly, they must both be in the closed position for the lamp L to light. We can state the behaviour of this circuit in various ways. For example, we can use plain English:

The lamp lights if we close both switches.

This can be expressed a little more concisely if we use symbols:

If A and B are true, then L is true.

By using a special kind of notation used in Boolean algebra, we can condense still further:

A.B=L

This is a Boolean statement (not an equation) and means:

If A and B THEN L

The dot between the two letters on the left implies they are connected by the AND function and the '=' sign simply means the word THEN. (The dot is sometimes omitted.) It is a highly symbolised form, difficult to grasp at first but extremely useful as a shorthand, so it is worth more explanation.

A switch or a lamp can only be in one of two possible states, either ON or OFF (operated or not operated). It is not too important what we call these states and we could equally well use any other pair of opposites like TRUE and FALSE or even 1 and 0. Thus the Boolean statement A.B=L could be

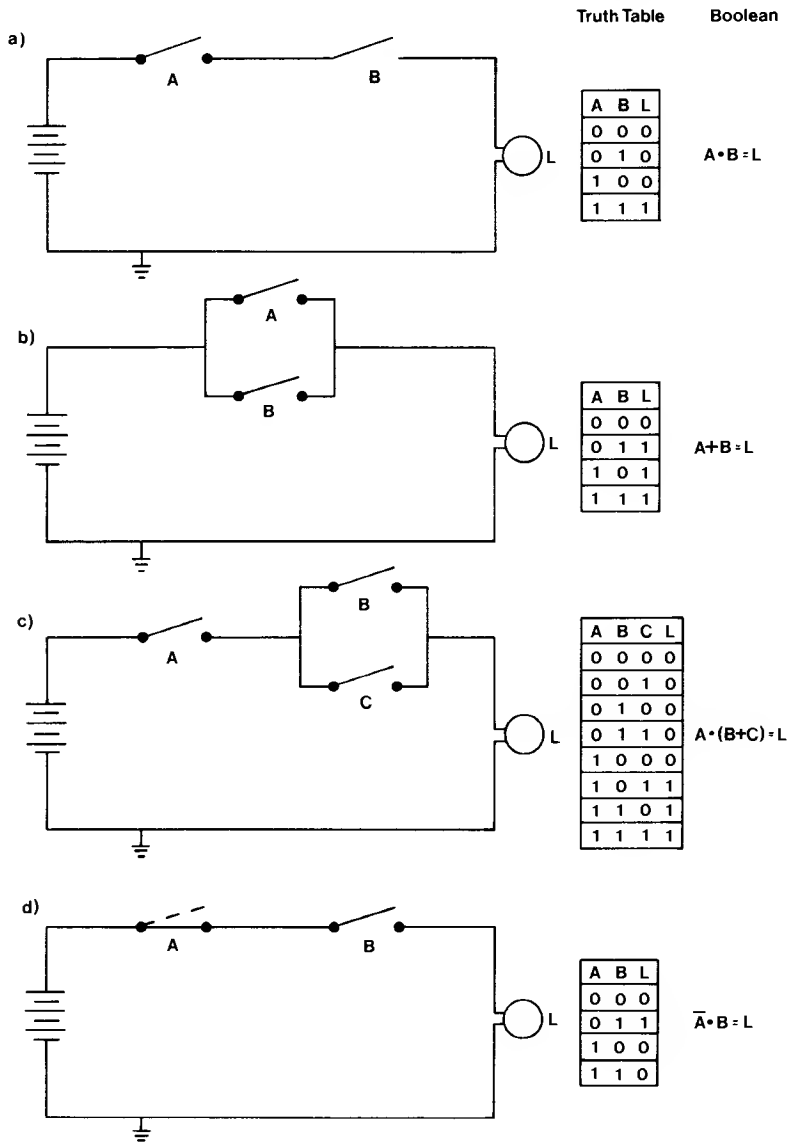


Fig. 8.1. Simple switch arrangements.

taken to mean 'If A and B are 1 THEN L is 1'. The use of 1 and 0 is very handy because it can be directly related to binary arithmetic.

Leaving Boolean for the moment and returning to Fig. 8.1(a), the essential point to grasp is that the switches are simulating the AND function. Another method of defining the action of this (and other switching circuits) is by means of the truth table. Since there are two switches, there must be four combinations. Using 0 for OFF and 1 for ON, the truth table shows there are three combinations which keep the lamp off

and only one which make it come on. It is tempting to sneer at such a truth table because it only appears to be stating the 'obvious'. Not all truth tables are as obvious as this. Writing out the truth table of a logic problem can often show up unexpected combinations.

The OR function

Figure 8.1(b) shows that the switches are now in parallel. In ordinary English, we could state:

> The lamp lights if switch A or B is operated.

Instead of an AND, we now have an OR function. We can express this in Boolean as follows:

A+B=L

Note carefully that the '+' sign means OR (not 'add'). You will also notice that the truth table shows three combinations for the lamp to be on and only one for off. The last line in the truth table is the AND function so there appears to be a slight overlap here between the OR and the AND. In fact, to be strictly accurate, this should be called the INCLUSIVE-OR function because it includes the AND function as well. However, it is the norm, when speaking of the OR function, to assume the inclusive form. It is worth mentioning at this point that there is a special form known as the EXCLUSIVE-OR which, as its name implies, excludes the AND function. That is to say, the last combination of two '1's gives a 0 instead of a 1.

AND and OR combinations

Figure 8.1(c) shows how one series and two parallel switches. Using ordinary English, we could state:

> The lamp lights if switch A is operated and either B or C is also operated.

This can be expressed in Boolean as follows:

A.(B+C)=L

Note the OR function is within brackets and that the A dot is outside in order to denote the AND function. Fortunately, brackets can be manipulated in Boolean just the same as in normal algebra so the expression can be expanded as follows:

A.B+A.C=L

This can be interpreted as 'The lamp lights if A and B are operated OR A and C are operated'.

Reverse action switches

Some switches, particularly those which are operated by relays, often work

in reverse. That is to say, the switch is normally in the closed position but becomes open when operated. Figure 8.1(d) shows that switch A is normally closed and switch B is normally open. Using ordinary English we could state:

The lamp will light if we operate switch B but do not operate switch A.

This can be expressed very neatly in Boolean:

$\bar{A}.B=L$

The bar over the top of A means 'not A' and is known as the negater bar and is a useful and concise way of indicating reverse action. If an input terminal on a logic chip is labelled, say, RESET then it is assumed that a HIGH level (logic 1) is required to cause reset action. Conversely, if it is marked $\overline{\text{RESET}}$, a LOW level would be required.

Common ground requirements

The circuits shown in Fig. 8 were only intended to introduce preliminary switching ideas. Practical logic circuits still use 'switch' action but obviously the switches have no moving parts. They are simply circuits which can change state (from HIGH to LOW) with respect to a common ground line. It can be seen from Fig. 8 that both sides of each switch are above ground level. This method of switching would be an awful nuisance in computer logic systems trying to work at high speed.

We shall now go on to examine some of the logic chips available in the so-called 'TTL family'.

Logic levels and TTL

Logic chips contain circuits which respond to, or deliver, one of two possible voltage levels. The TTL family of chips has set a common standard (see Fig. 8.2). All members of the TTL family (there are over 300 different chips) have type numbers beginning with 74 or 74LS. The LS prefix denotes Low-power Schockty and although similar in logic function they consume less current and are faster. ('Low-power Schockty' means that the transistors used have a special diode between the connector and base.) LS is now recommended for general use in favour of the traditional 'standard' TTL.

Logic 1 (also known as a HIGH) = any voltage within the range +2.8 V to 5 V.
Logic 0 (also known as a LOW) = any voltage within the range 0 V to +0.8 V.

Any voltage in between is called a 'bad level' and will lead to indeterminate results. Bad levels are usually caused by an output over load *pulling up* or

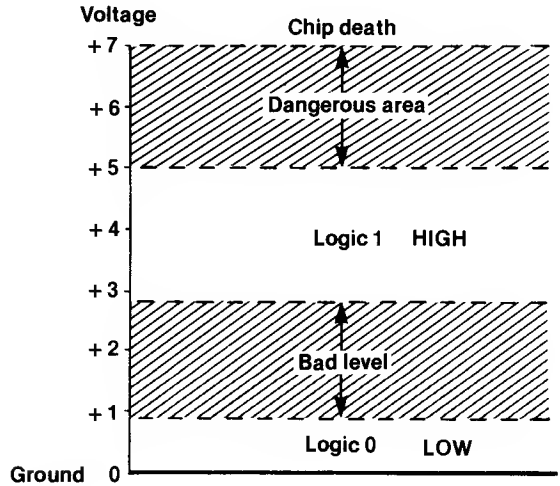


Fig. 8.2. TTL logic levels.

dragging down the voltage. Testing for HIGHs and LOWs at various points in the system can be done with a voltmeter although one of the various makes of 'logic probe' displaying either a red or green light is more convenient and less hazardous.

Logic gates

A gate is essentially a logic-operated switch with one output and one or more inputs. The combination of logic voltages on the inputs determines the output state. Although the function of a gate can be described in words, a truth table – with all possible input combinations – is concise and unequivocal. Figure 8.2 shows the six common gates in their most popular diagrammatic form, together with the corresponding truth tables. The inverter is not worth a truth table.

Notes on Fig. 8.2:

- *The AND gate:* output 1 only if all inputs 1.
- *The OR gate:* output 1 only if one or more inputs are 1.
- *The NAND gate:* output 0 only if all inputs are 1.
- *The NOR gate:* output 0 only if one or more inputs are 1.
- *The INVERTER:* output is reverse of input.
- *The EXCLUSIVE OR:* output 1 only if inputs are different. Examination of the truth table reveals that it is similar to the OR but 'excludes' the bottom AND line.

Although only two inputs are shown at each gate in Fig. 8.3; the TTL family includes gates with as many as eight inputs. The two common chips are the 7400 quad NAND and the 7404 hex inverter. The pin connections for these appear in Fig. 8.3. The power supply to the chips are marked Vcc (+5 V)

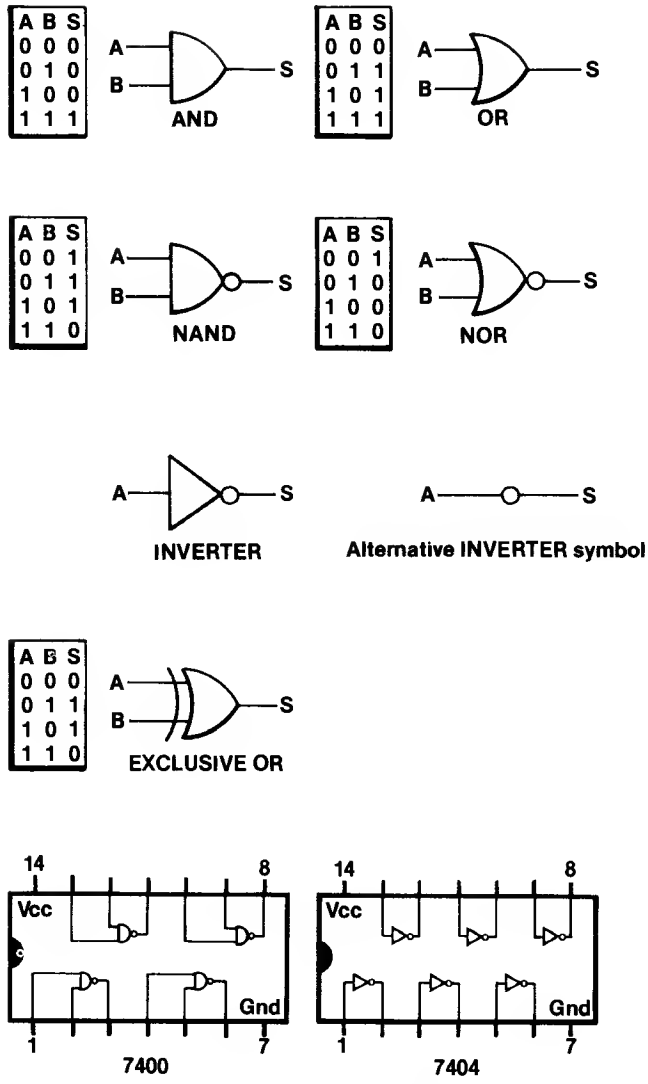


Fig. 8.3. The six primitive gates.

and Gnd (0 V) pin marked Gnd. Chip pin-out diagrams are always drawn looking down on to the top of the chip.

Active levels and negation

More complex chips such as decoders, buffers, counters, etc., do not always recognise a 1 or HIGH state as being in some way superior to a 0 or LOW state. Any input terminal which is immune to a 1 but activated by 0 is said to be active-LOW. such terminals are indicated either by (a) a bar over the terminal label such as $\overline{C}$ or clock (the bar is a Boolean symbol for negation – for example, $\overline{A}$ is the opposite state to A) or (b) a small circle or 'bubble'.

Dominance of NAND and INVERTER gates

TTL logic is based on the NAND and INVERTER, the other three gates tend to be under-used and therefore not so readily available. There are three reasons for their dominance:

(1) Many of the more complex chips are gated on by a LOW rather than a HIGH in order to minimise standby current.
(2) The internal circuitry of TTL gates is such that NAND and INVERTER functions arise more naturally and require less components.
(3) Combinations of NAND and INVERTER can be arranged to simulate AND and OR gates. An AND is an inverted output NAND. An OR is a NAND with all inputs inverted. Even the INVERTER is not strictly essential because a NAND, with all inputs strapped or held permanently at 1, behaves as an INVERTER.

Use of gates

Traditionally, the study of logic has leant heavily on a branch of mathematics known as Boolean Algebra. It is both a useful shorthand and a powerful tool in the mathematical analysis of logic. Boolean is still useful but, for the home enthusiast, the availability of complex integrated circuits has lessened the need to design and construct systems from an assortment of gates, so time spent on studying the special algebra may not always be justified.

The main use (now) for logic gates is to 'glue' together the more complex chips which may be incompatible in some way. For instance, one chip may deliver a 1 where a 0 is needed, meriting an inverter in between. Another possibility is the need to enable a chip only if 'something' else is at 1. Figure 8.4 shows some of the switching arrangements using gates.

Figure 8.4(a) shows an AND gate simulating a series switch in the data path. A serial data stream entering can only pass through the switch if the control C is HIGH.

If a NAND gate is used, as in Fig. 8.4(b), an inverter is needed. Without the inverter, the serial data stream would still pass if C is held HIGH but would be in inverted form (called the 'one's or logical complement').

Figure 8.4(c) shows how it is possible to enable a chip providing both A and B inputs are held HIGH. Note that the example chip is marked $\overline{CE}$ (not chip enable) which is convenient for the LOW output from the NAND. Remember that the bar over the CE label is an alternative to the bubble.

The exclusive-OR gate in Fig. 8.4(d) provides an easy way of controlling the phase of the output data. If the control C is held LOW, the output data stream is a replica of the input. If C is held HIGH, the output data is an inverted version of the input.

Figure 8.4(e) is a simulation of a single pole, double throw switch whereby the serial data stream can be diverted to either data out 1 or data out 2, depending on the state of the control C. If C is held HIGH, the data emerges from the bottom gate but from the top gate if held LOW.

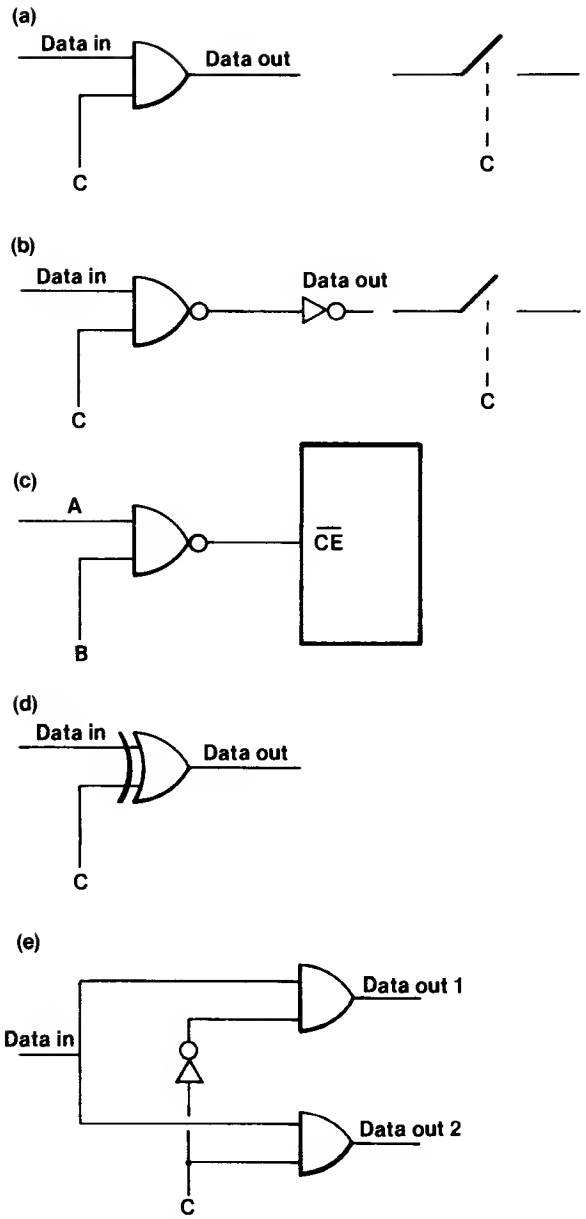


Fig. 8.4. Uses of simple gates.

INVERTERs would be needed at the outputs if NANDs were used instead of ANDs.

Flip-flops

Logic gates deliver an output state, depending on the present input

conditions. They are combinatorial devices, acting in real time and capable of analysis by simple Boolean algebra. Flip-flops are in an entirely different class because their present state depends on some event (usually a logic pulse) which occurred in the past. From this, it should be easy to conclude that flip-flops have the ability to memorise. But they can't memorise much. In fact, one flip-flop can only store a single bit so we would need eight of them to store one byte of data. A flip-flop which is storing a 1 is said to be set; if it is storing a 0, it is said to be reset. The four varieties of flip-flop in common use are shown in Fig. 8.5.

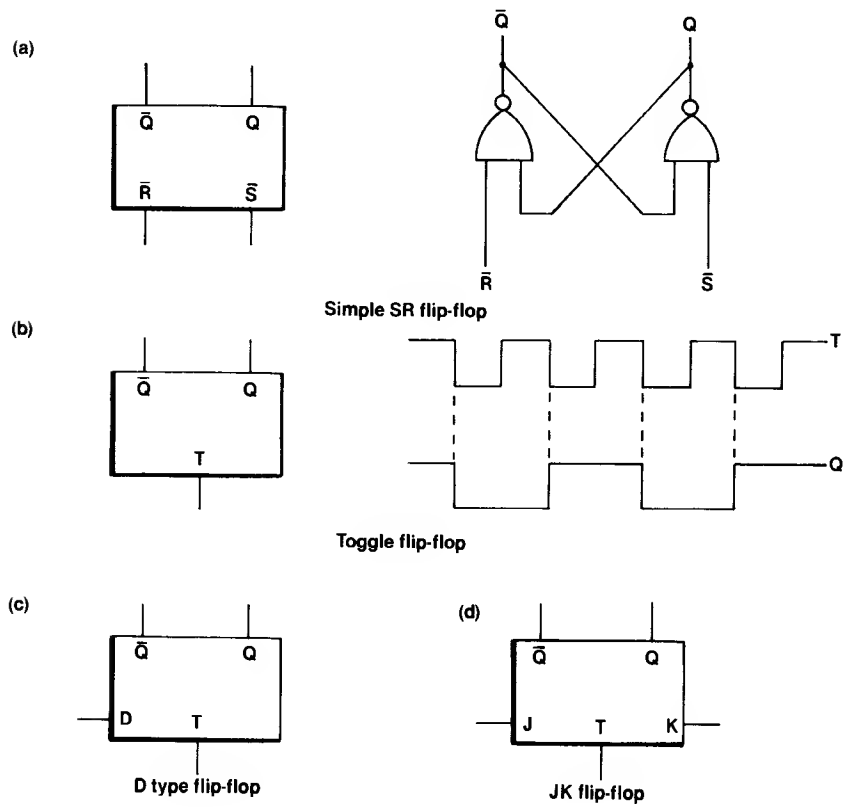


Fig. 8.5. Types of flip-flop.

The SR flip-flop. The logic symbol shows it to be a four-terminal black box. The output state is available at the 'Q' terminal which is set or reset by a negative-going pulse on S or R respectively. The term negative-going means a sudden drop in voltage from HIGH to LOW. It is important to realise that, although a transition from HIGH to LOW is required, it is not necessary to maintain the LOW state. In fact, the easiest way to try it out would be momentarily to touch the S terminal with a grounded wire. If it is already in the set state, nothing will happen. If it is in the reset state, the Q

terminal will go from 0 to 1 and remain in the new state until you flick the R terminal. The action is similar to the push-on/push-off switch found on table lights – it memorises the last order.

When a RS flip-flop is needed, it is customary to 'make' one from two cross-strapped NANDs (this only takes half a 7400). As a 'free gift', the Q terminal is always in the opposite state to $\overline{Q}$. Knowledge of this can often save an inverter.

The T flip-flop. This is often called a 'toggle' because every negative-going edge of a pulse on T will change the state at Q – it toggles the state backwards and forwards. The waveforms shown on Fig. 8.4(b) indicate that a continuous pulse of frequency f applied to T causes an output frequency of f/2, illustrating its primary use as a frequency divide-by-two stage. The Q output will be at the same half-frequency as Q but in the opposite phase. Direct set and reset terminals, which override T, may also be present in some types.

The D flip-flop. The D stands for 'Data'. The state at Q is oblivious to the D state until the trigger pulse arrives at T. When the negative going edge of the trigger arrives, the state of D (at that time) is passed (latched) into the flip-flop. In other words, the Q state is always the state which D was, prior to the arrival of the trigger. The 7475 is a quad D-type latch, containing four identical D flip-flops. Two of these can be used to latch in a byte of data.

The JK flip-flop. This is a versatile breed of flip-flop, shown in Fig. 8.4(d). The logic state on the J,K terminals decide the eventual state of Q after the next trigger pulse on T. The action is best described with the aid of the following truth table:

J	K	State of Q after next trigger
0	0	No change
0	1	Reset (Q=0)
1	0	Set (Q=0)
1	1	Change

Note that when J and K are both 0, the flip-flop is paralysed, unable to respond to any triggers.
If J and K are both held at 1, a trigger will always change the state. In other words, this perm of J,K transforms it to a T flip-flop.
If J is joined to K by an inverter, it is transformed to a D flip-flop, the J terminal acting as a D.
From this, it is easy to see why the JK flip-flop was described as versatile.

Finally, it should be mentioned that some diagrams will choose different

labels for the trigger terminal. The terminal we have marked T may, in some diagrams, be marked clock or just C.

Wired-OR and tristate outputs

A microprocessor system is based on the common bus. The output data from RAM, ROM, etc., are all wired in parallel across the same wires. It is important that such devices in the disabled state are effectively disconnected from the common bus. Normal TTL logic allows inputs to be connected together but under no circumstances must outputs be connected together unless they are of the class known as open-collector. Figure 8.6(a) shows the idea behind wired-OR connections.

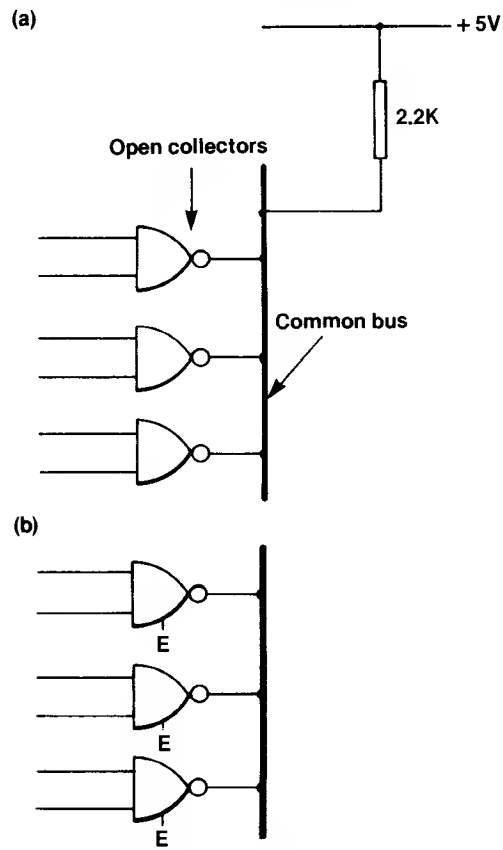


Fig. 8.6. Wired OR and tristate.

The output stage of normal TTL consists of two transistors in series across the 5 V supply (known as totem pole) with the gate output emerging from the centre point. The top transistor is missing in open-collector types and the feed to the 5 V line must come from an external pull-up resistor. This allows several outputs to be connected together, providing they all

share the same pull-up resistor. Many of the popular chips in the TTL family are available in both standard and open-collector versions.

Although wired-OR connections are useful in odd places, the solution is too messy for computer bus work. The alternative, and cleaner, solution is to provide chips with tristate outputs as shown in Fig. 8.6(b). An extra transistor is built into each output line, acting as a series switch and turned on or off by the enable terminal. When the chip is disabled, the outputs are effectively removed from the bus. The TTL chips offering tristate outputs are normally more complex than simple gates. RAM and ROM chips are almost always tristate.

Mechanical switches

Some disconcerting effects can occur if logic voltages are applied by means of an ordinary mechanical switch, particularly if the terminal supplied expects a single pulse. Due to the natural resonance of the operating spring, switches bounce backwards and forwards several times before coming to rest in the final position. The evil is called switch-bounce and can be overcome by either of the following two methods:

(1) Using an SR flip-flop and a single pole two-way switch as shown in Fig. 8.7. The flip-flop can be fashioned from the two strapped-NANDs previously described.

(2) Using software, incorporating a few milliseconds delay before 'reading' the state of the switch.

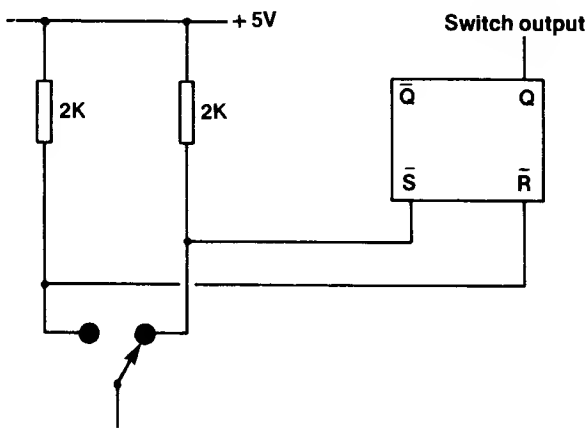


Fig. 8.7. Switch de-bounce circuit.

Driving lamps and relays

Small lamps are popular for displaying logic states. All lamps take current which can be ill afforded in logic work. Incandescent filament lamps are sluggish and take 50 mA or more. Neons take negligible current but require about 80 volts before they emit the characteristic red glow. This leaves the

light-emitting diode (LED) as the only serious contender. They give a reasonable light with about 5 mA and only drop about 1.2 volts. They must always be fed via a series resistor somewhere in the chain in order to ensure current, rather than voltage, drive. They are best driven from the output of an inverter as shown in Fig. 8.8.

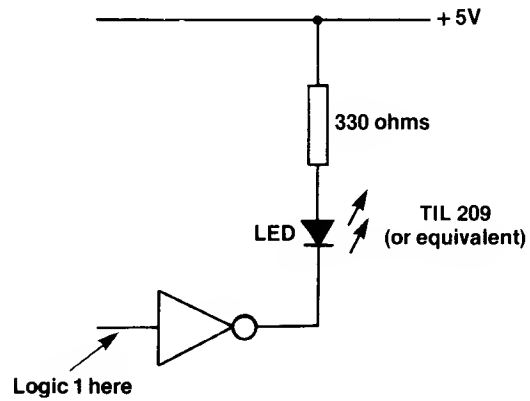


Fig. 8.8. Feeding a LED.

The LED lights when a 1 is applied to the inverter input. The inverter output then drops to near ground, completing the circuit through the LED. The output of the inverter is said to be sinking the LED current to ground.

Devices which require current in excess of 20 mA or voltages in excess of 5 V cannot be driven from logic circuits without help. This help can be supplied by the familiar electromagnetic relay, the opto-isolator or a combination of both. Figure 8.9 shows some arrangements.

In spite of the glamour associated with the semiconductor age, there are still uses for the traditional electromagnetic relay. Design methods have improved and the modern forms are efficient, physically small and take relatively low currents. Although no different in principle, the variant known as the reed relay, shown in Fig. 8.9(c), is common-place in modern interface circuitry. The operating contacts are enclosed within a glass tube filled with inert gas, which prevents the build-up of oxidation products. Because of this, the contact life is much higher than in the traditional open-contact relay. The operating coil is a separate component slipped over the tube and therefore can cater for a variety of current and impedance requirements.

Relays fulfil two primary requirements of the power interface:

(a) They allow the weak logic output from the computer to control high power.
(b) They electrically isolate the computer from high voltage circuits.

They must never be used without a reverse diode across the operat-

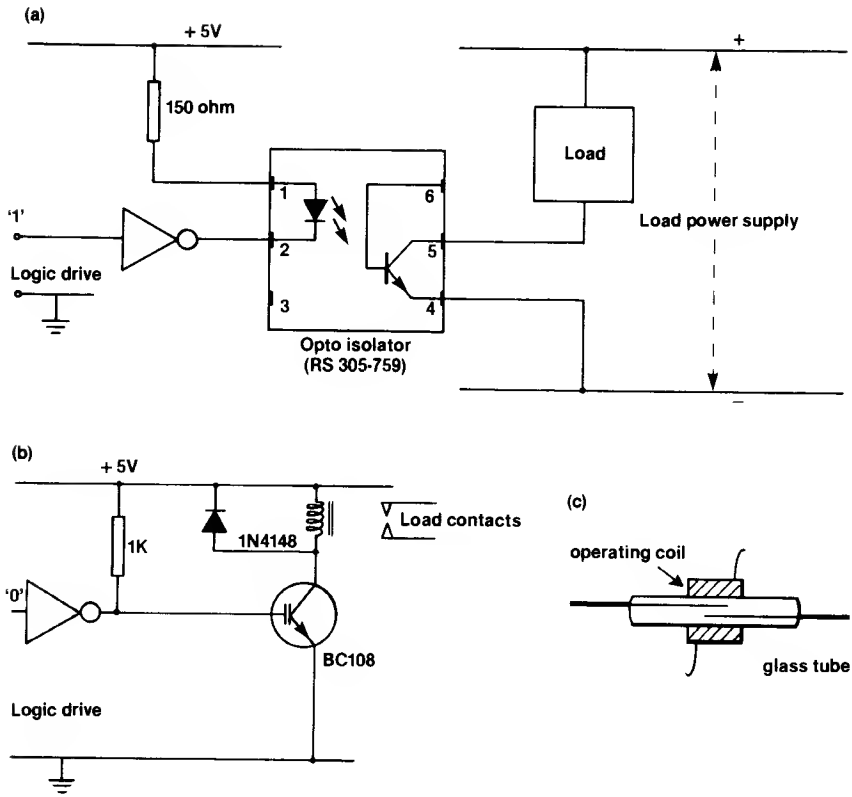


Fig. 8.9. Relays and opto-isolator drives.

ing coil. The diode safeguards the logic circuits from induced voltages which appear when the current is interrupted. Figure 8.9(b) shows a typical drive arrangement, using a common npn transistor as a current amplifier. The transistor conducts through the operating coil of the relay. The 1K resistor supplies the requisite base current. The presence of the inverter gate means that the transistor conducts on a logic 0 input and switches off on a logic 1. This is a case of an active-low drive causing a back-to-front action. If this is undesirable, the remedy is to insert an extra series inverter to bring it right again or substitute a non-inverting buffer gate. In either case, some form of logic gate is desirable in home constructed projects rather than a direct raw feed from the computer output port. Gates are cheap, computers aren't!

The opto-isolator is another popular component in interface work. Like the electromagnetic relay, the objective is to isolate electrically the computer from any high power/voltage/current components. In fact, the only connection is via the light emitted from a small LED falling on the base of a light-sensitive transistor. They are available singly as 6-pin chips, with the diode and transistor buried within the silicon. A typical circuit

using a single opto-isolator is shown in Fig. 8.9(a). The RS 305759 is only one of the many types available in the catalogues.

The box marked 'load' is a blanket term covering any contraption driven by the isolator. In all probability, this will include yet another transistor because the opto-isolator introduces a power 'gain' of less than unity (typically 0.2). To convert the loss into a gain, some opto-isolators incorporate two transistors and are classified as Darlington-connected. Some chips are available which contain four independent opto-isolators so two of these could handle the output from an 8-bit port.

Schmitt triggers

When the logic state changes from 0 to 1, or vice versa, logic chips expect the change to be rapid. In other words, the waveform should display, as far as possible, straight-sided pulses. If the input changes are sluggish, the behaviour could be impaired, particularly for clock-type inputs . If the input is obtained from the output of another logic circuit and the wiring between the two is not too long, there is no problem. However, if the input is obtained from an analogue, or 'home-made' source, the waveform is probably suspect and must be cleaned up before qualifying as a legitimate gate input. The 74 logic series has the answer in the form of the schmitt trigger, a standard circuit which accepts a poor pulse shape and transforms it into a steep-sided version. Figure 8.10 shows the gate symbol with typical input and output waveforms. The 7414 is a hex schmitt inverter, performing in the same way as a normal inverter but accepts poor waveforms. The schmitt does not protect against voltages which are out of range. It offers waveform but not voltage protection.

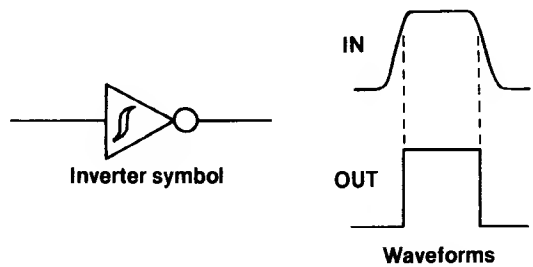


Fig. 8.10. The Schmitt trigger.

With reference to the mention of 'long' wires, it is worth pointing out that distributed capacity across wires, or between wires and ground, is often a cause of weird faults. It is sometimes a source of complaint that manufacturers of peripheral equipment appear to be miserly in the length of connecting cable supplied. In all fairness, this is not always due to penny-pinching. It is simply a wise precaution to avoid complaints of erratic behaviour which might arise if the cable length were increased. Apart

from distributed capacity, the longer the wire, the more chance of picking up stray induced voltages.

Timer chips

It must be admitted that discussion of these chips is a little out of place here. The 555 timer chip is not strictly a logic circuit although, if operated from a +5V supply, it accepts and delivers reasonable TTL voltage states. It is versatile, very low-priced, and easy to use. We are concerned here only with its use as a hardware timer, i.e. a device which, on receipt of a single narrow pulse, delivers an output HIGH state for a certain time before reverting to the quiescent LOW state automatically. Figure 8.11(a) shows the pin connections, wiring and waveforms.

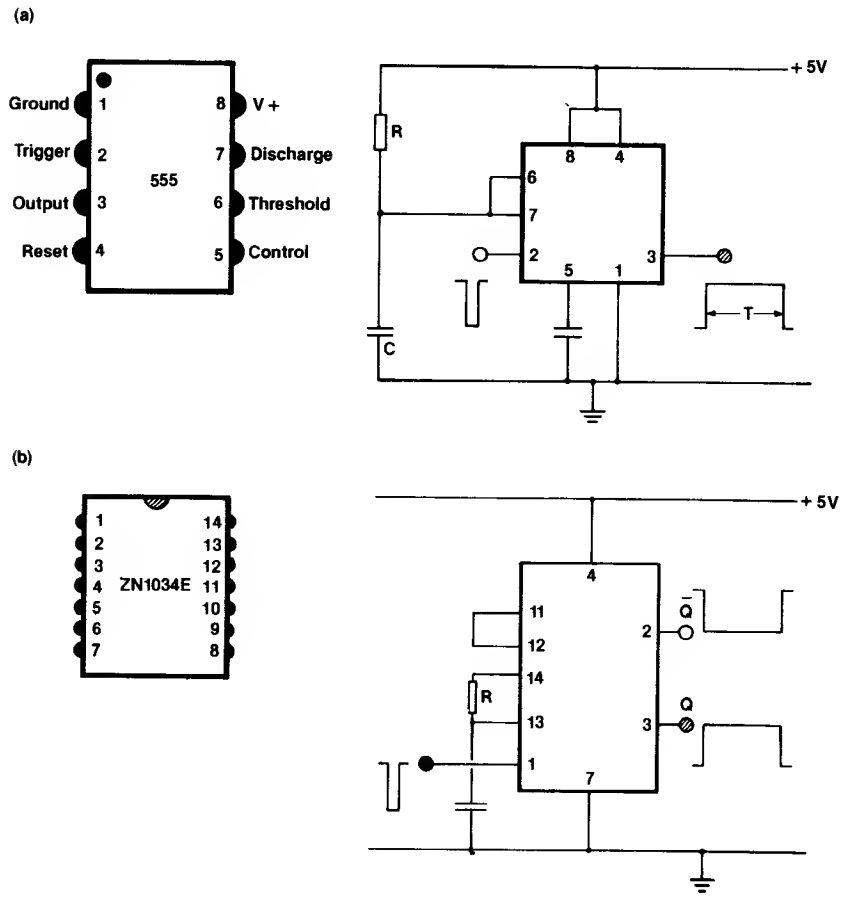


Fig. 8.11. The 555 and ZN1034E timers.

The 555 is ideal for cases where a single pulse from the computer can turn on a device (electric motor, perhaps) but is not required to stop under computer control. It is realised, of course, that the Commodore 64 includes

programmable timer facilities so the job could have been entirely software-controlled without a 555. However, it is good to be aware of alternative possibilities.

The output pulse-width, which determines the ON time of the device, is dependent on the value of C and R according to the following formula:

$$T = 1.1\ CR$$ (where C is in μF and R is in megohms)

For example, if C = 0.1 μF and R = 100K, the ON time will be 0.011 seconds. The figures illustrate that the 555 is not generally suitable for long time periods. It is not recommended to use R values greater than 1M, and capacities of the order of some microfarads means using electrolytics with wide tolerances. For periods over several seconds up to minutes, it is better to use one of the more sophisticated timers such as the ZN1034E shown in Fig. 8.11(b). The timing formula is:

$$T = 2736\ CR$$ (where C is in μF and R in megohms)

The multiplication factor 2736 is achieved by an internal 12-bit binary counter allowing time periods up to an hour or more. A useful feature is the provision of two complementary outputs, marked Q and $\overline{Q}$ in the diagram. It is a complex 14-pin chip with some of the pins allocated to external calibration resistors but only the simplified wiring is shown. To utilise the full potential, it is worth sending for detailed data sheets.

Decoders

A decoder will have several outputs but only one selected output can be activated at a time. The particular output depends on the specific combination applied to the selection input. Three select terminals can provide only eight different combinations of binary digits, the rule being:

Number of combinations of n bits = 2n

For example, to select any one of sixteen outputs requires four select inputs. There is a wide range of decoders in the TTL series. In addition to the select inputs, there will be one or more enabling inputs, allowing decoders to be linked together. Some of these may be active-high and some active-low. It is important to realise that all enable inputs must be activated before the chip becomes 'live'.

Demultiplexers

A demultiplexer routes serial input data to one particular output line and is the logic equivalent to a single pole multiway switch. Like the decoder, the particular output selected depends on the combination supplied to the select terminals. In fact, a decoder with enable inputs can be used as a demultiplexer by feeding the data to one which is active-high.

Multiplexers

These are mirror images of demultiplexers. They route any one of many input data sources to a single output line. The particular input source depends on the combination applied to the select inputs. The usual enable terminals will be present in most TTL chips.

Encoders

An encoder delivers a particular binary pattern on the output terminals. depending on which of the many input lines is activated. For example, there could be ten inputs, each capable of producing a unique four-bit pattern on the outputs and acting as a decimal to binary encoder. Most microcomputer keyboards are decoded by scanning software but some of the more expensive types are hardware encoded.

Counters

A counter is essentially a device which delivers an output binary pattern which changes on receipt of each input pulse. The TTL range offers a wide variety of counters. They may be classified as follows:

Binary counters. The input pulses cause the four-bit output to progress from 0000 to 1111 in a simple binary sequence. The pulse starts the count again at 0000.

BCD counters. The input pulses cause the four-bit output to progress from 0000 to 1001 (0 to 9 decimal). The tenth pulse starts the count again at 0000.

All counters will be supplied with a reset-to-zero input and most supply a terminal which emits a pulse when the count goes over the top to 0000. This is useful for cascading the output of one counter to the input of another. Two binary counters in cascade would then handle counts up to 1111 1111 (255 decimal) and two BCD counters up to 1001 1001 (99 decimal). There is another classification according to the direction of count. For example, those described above are up-counters but some varieties can be persuaded to down-count. For example, a four-bit binary down-counter has 1111 as the 'reset' state and decreases on each input pulse towards 0000. Down-counters are not supplied as such but some of the more sophisticated varieties have a control terminal which can be maintained HIGH for up-count and LOW for down-count. It is worth mentioning that an ordinary up-counter can be turned into a down-counter by inverting the outputs.

Shift-registers

It is self-evident that a shift-register shifts but, as with counters, there are generic variants depending on the direction of shift (left or right) and whether the initialised data is applied serially or in parallel. They will all have a 'shift' terminal (marked T or Clock). Every pulse on T shifts the contents one place, bits being pushed out at one end.

Parallel-in-parallel-outs, known as *PIPOs*, accept parallel data on the

four inputs, and data is available on the four output lines after the shift pulses have ended. The new input data is only let in to the register when an enabling level is applied to the appropriate terminal.

Parallel-in-serial-outs, known as *PISOs*, are similar to above but the output data can only be obtained a bit at a time on the serial output line.

Whatever other facilities they possess, shift-registers will always have serial-in and serial-out terminals. Many varieties exist in the TTL range. Some handle 8-bits and some can shift left or right depending on the state of a control terminal. The most obvious use for shift-registers is for parallel to serial or serial to parallel conversion.

Buffer registers

A buffer is a temporary holding register for data, the contents of which are subject to a latching pulse. Typically, there will be four data inputs, four data outputs and a terminal which is used to latch in the new data. Data variations at the input are 'unseen' until a latching pulse is applied when the current data overwrites the old. Some buffers have tristate outputs and are bi-directional.

Expansion ports

The expansion port on the Commodore 64 is the 44-pin socket at the back of the machine. For reasons of space alone, we will not attempt to dissect the port in any detail because pin connections and definitions are well explained in the Commodore Programmer's Reference Guide. It is hoped that our previous treatment of TTL logic will persuade readers to further their studies so that eventually they can design and connect their own expansion systems.

Expansion ports are normally intended for increasing existing facilities, particularly the amount of RAM and ROM. In view of this, it is understandable that the majority of the socket pins are tapped from the microprocessor address, data and control buses. It should be pointed out that extra RAM (or ROM) hooked on to the expansion port cannot be used to increase the total directly addressable memory space. For example, hooking on another 64K of RAM will not turn your computer into a 128K RAM because you cannot have access to all of it at once because the upper limit of 64K is imposed by the 16-bit address bus. All 16 address pins, A0 to A15, the 8 data pins, D0 to D7 and various control pins are available at the expansion port. There are also several ground and +5 volt lines although the current which can safely be drawn from them is minimal. It is not a good idea to rely too much on these voltages and they should not be regarded as a free supply source for all external equipment. To ease the problem of decoding sections of the address bus, special lines are provided on the expansion port.

ROML is an active-low decoded line for the address $8000, which means that only the least significant three digits need be decoded to access particular locations from $8000 upwards. Although the label implies ROM usage, RAM can equally well be used.

ROMH is similar to the above but provides an active low decoded output for address $E000.

DMA is the active-low Direct Memory Access line and is normally in the HIGH state. When this line is taken low, the address bus, data bus and read/write line from the 6510 microprocessor are effectively disconnected; they are said to have entered the high-impedance state. This allows an entirely separate control system to take over the memory chips. this separate system can be an extra 6510A or even a different microprocessor altogether. Obviously both microprocessors can't be in control at the same time but some form of switch action on the DMA line can change control of the buses from one to the other.

DOTCLOCK makes the 8.18 MHz clock oscillator on the main circuit board available to the expansion bus. All timing required by the microprocessor and memory chips are ultimately derived from this source although, apart from the video dot clock, the frequency is divided down before use.

The user port

Unlike an expansion port, which you may remember is primarily intended for memory expansion, a user port is completely undedicated. The designer of the machine will have no knowledge of the eventual use to which it may be put. Because of this, a user port is comparatively unsophisticated. The Commodore 64 user port provides a set of eight pins, labelled PB0 to PB7, which can be used to control, or be controlled by, any logic operated devices. The port is actually the B side of one of the CIAs. As explained in Chapter 1, any of the eight lines can act as either an input or an output depending on the bits programmed in the Direction Register. Logic 1 defines the appropriate line as an output and Logic 0 as an input. The actual logic on lines programmed as outputs depends on the corresponding bits programmed in the Data Register. The logic on input lines, which will arrive from an outside source, sets the corresponding bits in the Port Data Register.

The Address Register is addressed at $DD03 (56579 decimal). For example, if the lines corresponding to bits 4, 5, 6 and 7 are to be outputs and the rest inputs, the initialisation would be as follows:

```
LDA #$F0
STA $DD03
```

Always use hex when working out bit patterns because decimal notation is

just not worth the effort. For example, we want the bit pattern 1111 0000 in the following example. This is directly convertible, almost on sight, to $F0. It requires a lot of fumbling around to find the decimal equivalent. Try it! It should come to 128+64+32+16=240.

Suppose, having defined these lines as outputs, we now wish to set bit 7 to the HIGH state and bits 4,5 and 6 to the LOW state. It is the Port Data Register which is now important (the Direction Register is not responsible for the actual logic state on the lines).

The Port Data Register is addressed at $DD01 (decimal 56577), so to set the lines as specified above, we use the following coding:

```
LDA #80
STA $DD01
```

$80 is 1000 0000 in binary so bit 7 will be set HIGH. The pattern in the least significant nybble does not really matter because the right-hand half was directed to behave as inputs. This means that these port lines can only be defined by external inputs and so will ignore attempts to alter the logic by programming the Port Data Register.

Apart from the eight data lines, the port supplies two special lines which are used for controlling, rather than defining, the data lines. They are often referred to as 'handshaking' lines because they can be used for carrying on a kind of dialogue between the computer (which is a rigidly synchronised system) and the external device (which is usually unsynchronised and always undisciplined). For example, it would be useless sending data to a printer faster than the cogwheels can turn. What is needed is a signal from the port which asks 'Are you READY?' and another signal from the device when it has ACCEPTed the data. The two control lines are labelled FLAG and PA2 have the following specification:

FLAG. This is an input. On receipt of a negative-going edge (a drop from HIGH state to LOW), a FLAG interrupt bit is set which can, if suitably programmed, cause an interrupt signal to pass along the IRQ line to the microprocessor (refer back to Chapter 2).

PA2. This will normally be programmed as an output when used for handshaking purposes. It is not, in the usual meaning of the term, a true control line because it is simply one of the data lines (PA2) 'borrowed' from the A side port of the same CIA. The direction register for the A side is located at $DD02 and the port data register at $DD00. Care must be taken when programming these registers for a particular PA2 behaviour, that only bit 2 is affected otherwise the serial bus action (which uses PA3 to PA7) will be disturbed. The safest way is to program by mask techniques, as described in Chapter 3.

The IEEE bus

Commodore have always been keen on the special standardised interface known as the IEEE bus so it is worth digressing a little in order to trace its history. In 1972, Hewlett-Packard cooperated with various USA and European bodies attempting to standardise some form of general purpose interface between computers and instruments. The design submitted by Hewlett-Packard was provisionally accepted by the IEEE (USA) in 1975. The full title of the agreed system was IEEE 488–1975 although the modified form by Hewlett-Packard was called the HP–IB. The specification contained elaborate detail, even down to the exact spacing and materials to be used in interface sockets. Needless to say, standards soon wilt under the onslaught of free market forces. Cost-effective 'improvements', while they may continue to preserve the original framework, tend to generate hybrids.

The IEEE protocol

Suppose we wish to use a computer to control, say, ten or more different peripherals, any of which must be on-line at the same time in case they are to be called. The first obstacle which arises is the number of sockets required, assuming one is used for each peripheral. Apart from the cost of supplying enough sockets to cover the future demand, there would still be the problem of finding enough space at the back of a microcomputer for fitting them. The IEEE bus idea overcomes this obstacle by employing only one interface plug at the back of the computer. All peripherals connect to this on what has become known as a 'daisy chain'.

Daisy chains

'Daisy chains' means using sets of interface cables with male and female sockets on one end and male on the other. One of these is then plugged into the computer and the other into the first peripheral. The next peripheral is fed from the female socket of the first cable and so on. Thus we may have ten or more peripherals all 'daisy chained' to each other but all fed from one computer. This is the answer to the socket problem. However, it is obvious that a second, and potentially more serious, obstacle now arises. How does the computer correspond with a particular peripheral to the exclusion of all the rest? As far as the bus wires within the connecting cable are concerned, the peripherals are all in parallel and consequently, indistinguishable from one another. This problem is overcome, partly by hardware and partly by software, in the following manner:

Standardised signals.

Every wire in the bus has a rigidly defined signal function as laid down by the IEEE protocol.

Device address.

Peripherals must have four switches, or movable jumper wires for

establishing a 4-bit device number, unique to that peripheral. The total number of devices on the bus must not exceed 16, including the computer itself. (Four switches have 16 combinations.) Once the switches have been set, the binary combination becomes the device address of the peripheral. For example, if the switches have been set to HIGH HIGH LOW HIGH (1101 in binary) the device address is $D or 13 in decimal. When commands are sent along the bus they are preceded by a device address. Peripherals with a different address would not recognise the command.

The secondary address. As stated earlier, the original objective of the IEEE bus system was to provide a general-purpose interface for the many digitally controlled instruments which began to flood the market in the seventies. Such instruments, which included frequency generators and analysers, voltage and current meters, etc., were becoming sophisticated and each capable of performing a wide variety of functions. To incorporate such instruments into an integrated computerised control system required a device address and also a 'secondary address'. The secondary address is regarded as the address of one particular function within the total repertoire of the peripheral. The secondary address would follow the device address at the head of a bus command. For example, a digital voltmeter might have the device number 7 but possibly eight or more secondary addresses to cover the ranges of measurement. A peripheral is limited to 31 secondary addresses, which is more than adequate for most devices. In fact, as far as ordinary computer peripherals are concerned (floppy disks, for example) two secondary addresses would be ample, one for READ and one for WRITE.

Listeners and talkers

The IEEE establishment thought it would be a good idea to employ nice homely terms for the bus. They decided to classify devices as either 'talkers' or 'listeners'. A *talker* is a device which can only send data to the bus. A *listener* is a device which can only receive data from the bus. A keyboard can only talk and a printer can only listen. Many devices can either talk or listen, depending on their present mode. For example, a floppy disk can both talk and listen. The computer has been rechristened a 'controller' and there can be only one of them on the bus.

Handshaking protocol

The complete specification for handshaking covers all possible eventualities and is an awe-inspiring read. The serial bus on the Commodore 64 employs a simplified form of IEEE bus. External devices addresses are within the brand 4 to 31, the lower addresses belong internally. Device address 4 or 5 will call up the VIC-1525 Graphic Printer and device 8 is the VIC-1541 disk drive. These addresses can be changed by internal adjustments if required. The full details of the serial bus are given in the Commodore Programmer's Reference Guide.

Summary

1. There are only two logic states, the '1' state and the '0' state. Alternatively, they can be called the HIGH and LOW states.
2. TTL logic uses a nominal 5 volt power supply, positive to ground.
3. The HIGH state is any voltage between about 2.8 and 5 volts.
4. The LOW state is any voltage between about 0.8 volts and ground.
5. Any voltages in between HIGH and LOW are known as bad states and are suspect.
6. Any input terminal requiring a HIGH to activate it is called an active high terminal.
7. Any input terminal requiring a LOW to activate it is called an active low terminal.
8. On a logic diagram, active low inputs are either recognised by a small bubble at the input or by a bar over the label.
9. Flip-flops are single bit storage cells which can rest, or be triggered into, the SET or RESET state.
10. Devices with tristate outputs can be either HIGH, LOW or in the high impedance 'dead' state.
11. Tristate devices are superior to wired-OR for connection to a paralleled bus system.
12. Relays must have reverse diodes across the operating coil if driven by TTL devices.
13. Schmitt triggers circuits are used to clean up bad edges on a pulse.
14. Decoders have many outputs, only one of which can be activated at any one time.
15. Multiplexers select one of the input signals for passage through to the output.
16. Demultiplexers select one of the outputs to receive the input signal.

Self test

8.1 State the main advantage of CMOS over TTL logic.

8.2 Write the Boolean statement corresponding to the following plain language statements:
(a) For the electric crane (C) to operate, switch A must be operated or switch B must not be operated.
(b) To fire the ballistic missile (M), switches A,B,C and D must all be operated but switch E must not be operated if switch F is operated.

8.3 Why are LS versions of the TTL family favoured?

8.4 A NOR gate gives a 1 out if none of the inputs are 1. True or false?

8.5 A NAND gate gives a 1 out if all inputs are 0. True or false?

8.6 If both inputs to an exclusive OR gate are 1, what is the output state?

8.7 If a NAND gate output is fed to an inverter, what is the equivalent logic gate?

8.8 A NOR gate has inverters wired to all inputs. What is the equivalent logic gate?

8.9 What is the alternative to tristate connections to a common bus?

8.10 Electromagnetic relays must have a diode connected across the operating coil. Why?

Appendix A
Binary and Hex

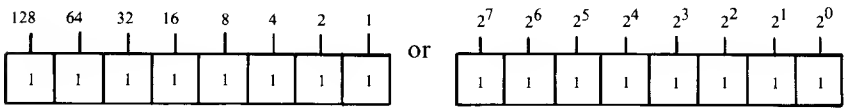
Binary

- Unless otherwise stated, numbers will be assumed to be decimal. Hex numbers are prefixed by $.
- To aid comprehension, strings of bits may be split into groups of four, but the space between groups is artificial.
- 'X' is used for 'don't care' bits and can mean 1 or 0.
- To 'flip' a bit means to change it from 1 to 0 or vice versa.

Unsigned binary system

Computer languages, whether entered in high level, assembly coding, or hexadecimal, are incomprehensible to the machine. All information is converted by the resident operating system to binary bits (1s and 0s).

All number systems, including the familiar decimal, rely on the relative position of digits to indicate their 'worth'. Each binary digit in a byte is twice the value of the bit to its right. In pure unsigned binary, the value of each binary 1 is shown below in both decimal equivalents and powers of two:



Examples:

1000 1001 = 137
1001 1111 = 159
1111 1111 = 255

Sometimes, the following tip is useful:

A string of all 1s = 2^n-1, where n = number of bits in the string.

Examples:

1111 = 2^4-1 = 15
1111 1111 1111 1111 = $2^{16}-1$ = 65535

It is advisable, but not essential, to memorise powers of two up to the first sixteen bit positions. It is convenient to consider them divided into low-byte and high-byte as follows:

Powers of 2																
High-byte								Low-byte								
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	n
(32768)	(16384)	(8192)	(4096)	(2048)	(1024)	(512)	(256)	(128)	(64)	(32)	(16)	(8)	(4)	(2)	(1)	2^n

Any binary number in the high-byte position is always 256 times its low-byte value. For example: 0000 1001 would be worth 9 if low-byte, but 256*9 = 2304 if high-byte. Remember, the 6502 always stores 16-bit data in consecutive memory addresses, low-byte first.

Hexadecimal notation (hex)

Hex uses the 16 characters 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F to describe a nibble (4 bits):

0000 = 0 0001 = 1 0010 = 2 0011 = 3
0100 = 4 0101 = 5 0110 = 6 0111 = 7
1000 = 8 1001 = 9 1010 = A 1011 = B
1100 = C 1101 = D 1110 = E 1111 = F

Two hex characters describe a byte. Some examples follow:

1111 0011 = F3 0001 1011 = 1B 1100 1101 = CD 0000 0001 = 01
1111 1111 1111 1111 = FFFF 1000 1100 1010 0111 = 8CA7

Hex arithmetic

Hex is based on powers of 16 so any character, depending on its position, must be multiplied by the appropriate power of 16 as follows:

16^3 =4096 16^2 =256 16^1 = 16 16^0 =1

	4096	256	16	1
Using H for hex character:	H	H	H	H

Examples:

$0032=(3*16)+2=50 $00FC=(15*16)+12 = 252
$00FF=(15*16)+15=255
$203E=(2*4096)+(3*16)+14=8254
$1111=4096+256+16+1=4369

Signed binary and two's complement

In order to represent both positive and negative numbers in a byte, the msb (bit 7) is reserved as the 'sign' bit.

The sign bit is 1 for negative and 0 for positive numbers. For example:

0XXX XXXX is positive and 1XXX XXXX is negative.

A negative number is said to be the *two's complement* of the equivalent positive and vice versa. There are two ways of obtaining the two's complement of a binary number:

(1) First flip all the bits and then add one. Ignore any carry out from msb end.
(2) Starting from the lsb, copy up to and including the first '1' then flip the remaining bits.

Examples:	Number	Two's complement
(+7)	0000 0111	1111 1001 (−7)
(+1)	0000 0001	1111 1111 (−1)
(−2)	1111 1110	0000 0010 (+2)

Method 1 can lead to errors when adding the 1, so method 2 is safer. The two's complement of decimal numbers is found by subtracting from 256.

Example: −1 = 1−256 = 255 = 1111 1111.

The two's complement of hex numbers is found by subtracting from &FF and adding 1.

Example: −3 = &FF−3 = &FC+1 = &FD

The largest positive number in a byte is +127 = 0111 1111 = &7F.
The largest negative number is −128 = 1000 0000 = &80.

Notes:
(a) The larger the negative number, the more binary 0s appear. In two's complement, everything is reversed, including the relative status of 1s and 0s.
(b) There are 128 positive and 128 negative numbers. The fact that zero is a positive number is the reason why there appears to be one more negative than positive (−128,+127).

Binary coded decimal (BCD)

Decimal numbers are awkward when expressed in binary, simply because base 10 and base 2 don't mix well. BCD is a code which sacrifices efficiency for decimal compatibility. A byte is divided into two 4-bit groups (nibbles). Each nibble is coded for numbers from 0 to 9, as follows:

BCD	Decimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9

The six groups from 1010 to 1111, which are used for the characters A to F in hex, are *illegal* in BCD. A single byte can hold decimal numbers in BCD form only in the range, 0 to 99.

Examples: 0001 0011=13 0000 0111=07 1001 1001=99

The *efficiency* of a code = (number of combinations used) ÷ (total combinations).

In pure binary, all combinations are used, so the efficiency is 100%. In BCD, only 10 combinations are used out of a total of 16 possible, so the efficiency is 10/16=63% approximately. When the efficiency within a byte is calculated, the loss in information content is worse - 100/256 which is not quite 40%.

Because of the inefficiency of BCD, its use is limited. However, a large proportion of digital instrumentation delivers, or expects to receive, information in BCD form. The 6502 microprocessor obligingly processes BCD arithmetic if the D flag in the processor register is set to 1. However, it is up to the programmer to ensure that the data entering the arithmetic area is free from illegal groups.

Appendix B
Kernal Subroutines

ACPTR
Function: get data from serial bus
Call address: $FFA5 or 65445 decimal
Parameter register: accumulator receives data byte
Preparation routines: TALK,TKSA
Example:

```
JSR ACPTR
```

CHKIN
Function: open input channel
Call address: $FFC6 or 65478 decimal
Parameter register: X must contain channel number
Preparation routines: OPEN
Example:

```
LDX #4
JSR CHKIN
```

CHKOUT
Function: open output channel
Call address: $FFC9 or 65481
Parameter register: X
Preparation routines: OPEN
Example:

```
LDX #2
JSR CHKOUT
```

CHRIN
Function: get character from input channel
Call address: $FFCF or 65487

Parameter register: accumulator receives data byte
Preparation routines: OPEN,CHKIN
Example:

```
JSR CHRIN
STA DATA
```

CHROUT
Function: send character to output channel
Call address: $FFD2 or 65490
Parameter register: accumulator must contain character code
Preparation routines: CHKOUT,OPEN
Example:

```
LDA #42
JSR CHROUT
```

CIOUT
Function: Send data over serial bus
Call address: $FFA8 or 65448 decimal
Parameter register: accumulator must contain data byte
Preparation routines: LISTEN and, if necessary, SECOND
Example:

```
JSR CIOUT
```

CINT
Function: initialise screen editor and VIC
Call address: $FF81 or 65409 decimal
Parameter register: not applicable
Preparation routines: not applicable
Example:

```
JSR CINT
```

CLALL
Function: close every file
Call address: $FFE7 or 65511 decimal
Parameter register: not applicable
Preparation routines: not applicable
Example:

```
JSR CLALL
```

CLOSE
Function: close a file
Call address: $FFC3 or 65475 decimal
Parameter register: accumulator must contain logical file number
Preparation routines: not applicable
Example:

```
LDA #3
JSR CLOSE
```

CLRCHN
Function: clear input/output channels
Call address: $FFCC or 65484 decimal
Parameter register: not applicable
Preparation routines: not applicable
Example:

```
JSR CLRCHN
```

GETIN
Function: get character from keyboard buffer
Call address: $FFE4 or 65508 decimal
Parameter register: accumulator receives data
Preparation routines: CHKIN,OPEN
Example:

```
BACK  JSR GETIN
      CMP #0
      BEQ BACK
```

IOBASE
Function: Set input/output memory page
Call address: $FFF3 or 65523 decimal
Parameter registers: X and Y. After calling, X and Y contain the low- and high-byte address respectively of the memory-mapped I/O devices.
Preparation routines: not applicable
Example:

```
JSR IOBASE
```

IOINIT
Function: initialise every input/output device
Call address: $FF84 or 65412 decimal
Parameter register: not applicable
Preparation routines: not applicable

Example:

```
JSR IOINIT
```

LISTEN
Function: cause serial bus device to listen
Call address: $FFB1 or 65457 decimal
Parameter register: accumulator must contain device number
Preparation routines: not applicable
Example:

```
LDA #6
JSR LISTEN
```

LOAD
Function: load data from device into RAM
Call address: $FFD5 or 65493 decimal
Parameter registers: accumulator must contain 0 (for load) or 1 (for verify). If input device has been opened with a secondary address 0, the X and Y registers must contain the low- and high-byte address respectively of the RAM starting address.
Preparation routines: SETLFS,SETNAM
Example:

```
LDA #0
JSR LOAD
```

MEMBOT
Function: sets bottom of RAM
Call address: $FF9C or 65436 decimal
Parameter registers: X and Y must contain the low- and high-byte respectively of the bottom address. (The normal default value is $0800 or 2048 decimal.)
Preparation routines: not applicable
Example:

```
CLC
JSR MEMBOT
```

MEMTOP
Function: set top of RAM
Call address: $FF99 or 65433 decimal
Parameter registers: X and Y must contain the low- and high-byte respectively of the top address
Preparation routines: not applicable

Example:

```
CLC
JSR MEMTOP
```

OPEN
Function: open a file
Call address: $FFC0 or 65472 decimal
Parameter register: not applicable
Preparation routines: SETLFS,SETNAM
Example:

```
JSR OPEN
```

PLOT
Function: Set location of cursor
Call address: $FFF0 or 65520 decimal
Parameter register: X and Y must contain the coordinates of the cursor position
Preparation routines: not applicable
Example:

```
LDX #13
LDY #20
CLC
JSR PLOT
```

RAMTAS
Function: execute memory test, initialise memory pointers and various workspaces
Call address: $FF87 or 65415 decimal
Parameter registers: not applicable
Preparation routines: not applicable
Example:

```
JSR RAMTAS
```

RDTIM
Function: reads time clock
Call address: $FFDE or 65502 decimal
Parameter register: accumulator, X and Y registers receive the high, middle and low bytes respectively of the current system clock
Preparation routines: not applicable
Example:

```
JSR RDTIM
```

```
STY CLOCK
STX CLOCK+1
STA CLOCK+2
```

READST
Function: read input/output status word
Call address: $FFB7 or 65463 decimal
Parameter register: accumulator receives the word
Preparation routine: not applicable
Example:

```
JSR READST
```

RESTOR
Function: restores all system vectors and BASIC routines and interrupts to their default states
Call address: $FF8A or 65418 decimal
Parameter register: not applicable
Preparation routines: not applicable
Example:

```
JSR RESTOR
```

SAVE
Function: Save memory on output device
Call address: $FFD8 or 65496 decimal
Register parameters: accumulator must have the indirect address of the start and X and Y must have the address of the end of the block to be saved
Preparation routines: SETLFS,SETNAM
Example:

```
LDX ENDLOW
LDY ENDHIGH
LDA START
JSR SAVE
```

SCNKEY
Function: scan keyboard and transfer character to buffer
Call address: $FF9F or 65439 decimal
Parameter register: not applicable
Preparation routine: IOINIT
Example:

```
SCAN  JSR SCNKEY
      JSR GETIN
```

```
CMP #$00
BEQ SCAN
```

SECOND
Function: send secondary address for LISTEN
Call address: $FF93 or 65427 decimal
Parameter registers: accumulator must contain the secondary address
Preparation routines: LISTEN
Example:

```
LDA #12
JSR SECOND
```

SETLFS
Function: set up a file
Call address: $FFBA or 65466 decimal
Parameter register: X must contain the file number; Y must contain the secondary address (command) but if there is no secondary address, Y must contain $FF; accumulator must contain the device number.
Preparation routines: not applicable
Example:

```
LDA #6
LDX #5
LDY #$FF
JSR SETLFS
```

SETNAM
Function: set up file name
Call address: $FFBD or 65469 decimal
Parameter registers: accumulator must contain length of file name; X and Y registers must contain low- and high-byte respectively of the address where the name is stored
Preparation routines: not applicable
Example:

```
LDX LOWNAME
LDY HIGHNAME
LDA LENGTH
JSR SETNAM
```

SETTIM
Function: set time clock in units of 1/60 seconds
Call address: $FFDB or 65499 decimal
Parameter registers: accumulator must contain high-byte, X must contain

high-byte, X must contain middle-byte and Y the low-byte
Preparation subroutines: not applicable
Example:

```
LDA #HIGHEST
LDX #LOWBYTE
LDY #HIGHBYTE
```

SETTMO
Function: set IEEE bus time-out flag
Call address: $FFA2 or 65442 decimal
Parameter register: accumulator must contain 0 in bit 7 if time-out is to be enabled but 1 if to be disabled
Preparation routines: not applicable
Example:

```
LDA #0
JSR SETTMO
```

or

```
LDA #$80
JSR SETTMO
```

STOP
Function: to check STOP key
Call address: $FFE1 or 65505
Parameter register: accumulator will receive a byte representing the last row of the scan if STOP is not found
Preparation routines: not applicable
Example:

```
JSR STOP
```

TALK
Function: cause a serial bus device to talk
Call address: $FFB4 or 65460 decimal
Parameter register: accumulator must contain device number
Preparation routines: not applicable
Example:

```
LDA #7
JSR TALK
```

TKSA
Function: send secondary address number to a device caused to talk
Call address: $FF96 or 65430 decimal

Parameter register: accumulator must contain secondary address number
Preparation routine: TALK
Example:

```
LDA #1
JSR TKSA
```

UDTIM
Function: update time clock
Call address: $FFEA or 65514 decimal
Parameter register: not applicable
Preparation routines: not applicable
Example:

```
JSR UDTIM
```

UNLSN
Function: command all devices to stop listening to the bus
Call address: $FFAE or 65454 decimal
Parameter register: not applicable
Preparation routines: not applicable
Example:

```
JSR UNLSN
```

UNTLK
Function: command all devices to stop talking to the bus
Call address: $FFAB or 65451
Parameter registers: not applicable
Preparation routines: not applicable
Example:

```
JSR UNTLK
```

Appendix C
6502/6510A Complete Instruction Set

Appendix C1
6502 Complete Instruction Set

ADC	Add with carry	A+M+C→A	NZCV

Address mode	Op-code	Bytes	Cycles
Immediate	$69	2	2
Zero-page	$65	2	3
Zero-page,X	$75	2	4
Absolute	$6D	3	4
Absolute,X	$7D	3	4 or 5
Absolute,Y	$79	3	4 or 5
(Indirect,X)	$61	2	6
(Indirect),Y	$71	2	5

AND	And with A	A and M→A	NZ

Address mode	Op-code	Bytes	Cycles
Immediate	$29	2	2
Zero-page	$25	2	3
Zero-page,X	$35	2	4
Absolute	$2D	3	4
Absolute,X	$3D	3	4 or 5
Absolute,Y	$39	3	4 or 5
(Indirect,X)	$21	2	6
(Indirect)			
(Indirect,X)	$21	2	6
(Indirect),Y	$31	2	5

ASL	Shift left	C←(7...0)←0	NZC

Address mode	Op-code	Bytes	Cycles
Accumulator	$0A	1	2
Zero-page	$06	2	5
Zero-page,X	$16	2	6
Absolute	$0E	3	6
Absolute,X	$1E	3	7

BCC	Branch if C=0	Flags unaltered

Address mode	Op-code	Bytes	Cycles
Relative	$90	2	2 or 3

BCS	Branch if C=1	Flags unaltered

Address mode	Op-code	Bytes	Cycles
Relative	$B0	2	2 or 3

BEQ	Branch if Z=1	Flags unaltered

Address mode	Op-code	Bytes	Cycles
Relative	$F0	2	2 or 3

BIT	A and M,M7→N,M6→V	Z,N,V

Address mode	Op-code	Bytes	Cycles
Zero-page	$24	2	3
Absolute	$2C	3	4

BMI	Branch if N=1	Flags unaltered

Address mode	Op-code	Bytes	Cycles
Relative	$30	2	2 or 3

BNE	Branch if Z=0	Flags unaltered

Address mode	Op-code	Bytes	Cycles
Relative	$D0	2	2 or 3

BPL	Branch if N=0	Flags unaltered

Address mode	Op-code	Bytes	Cycles
Relative	$10	2	2 or 3

BRK	Break PC+2	I flag=1

Address mode	Op-code	Bytes	Cycles
Implied	$00	1	7

BVC	Branch if V=0	Flags unaltered

Address mode	Op-code	Bytes	Cycles
Relative	$50	2	2 or 3

BVS	Branch if V=1	Flags unaltered

Address mode	Op-code	Bytes	Cycles
Relative	$70	2	2 or 3

CLC	Clear Carry	C flag=0

Address mode	Op-code	Bytes	Cycles
Implied	$18	1	2

CLD	Clear Decimal	D flag=0

Address mode	Op-code	Bytes	Cycles
Implied	$D8	1	2

CLI	Clear I mask	I flag=0

Address mode	Op-code	Bytes	Cycles
Implied	$58	1	2

CLV	Clear overflow	V flag=0

Address mode	Op-code	Bytes	Cycles
Implied	$B8	1	2

CMP	Compare A	A-M	NZC

Address mode	Op-code	Bytes	Cycles
Immediate	$C9	2	2
Zero-page	$C5	2	3
Zero-page,X	$D5	2	4
Absolute	$CD	3	4
Absolute,X	$DD	3	4 or 5
Absolute,Y	$D9	3	4 or 5
(Indirect,X)	$C1	2	6
(Indirect),Y	$D1	2	5 or 6

CPX	Compare X	X-M	NZC

Address mode	Op-code	Bytes	Cycles
Immediate	$E0	2	2
Zero-page	$E4	2	3
Absolute	$EC	3	4

CPY	Compare Y	Y-M	NZC

Address mode	Op-code	Bytes	Cycles
Immediate	$C0	2	2
Zero-page	$C4	2	3
Absolute	$CC	3	4

DEC	Decrement M	M−1→M	NZ

Address mode	Op-code	Bytes	Cycles
Zero-page	$C6	2	5
Zero-page,X	$D6	2	6
Absolute	$CE	3	6
Absolute,X	$DE	3	7

DEX	Decrement X	X−1→X	NZ

Address mode	Op-code	Bytes	Cycles
Implied	$CA	1	2

DEY	Decrement Y	Y−1→Y	NZ

Address mode	Op-code	Bytes	Cycles
Implied	$88	1	2

EOR	Exclusive-OR	AexcM→A	NZ

Address mode	Op-code	Bytes	Cycles
Immediate	$49	2	2
Zero-page	$45	2	3
Zero-page,X	$55	2	4
Absolute	$4D	3	4
Absolute,X	$5D	3	4 or 5
Absolute,Y	$59	3	4 or 5
(Indirect,X)	$41	2	6
(Indirect),Y	$51	2	5

INC	Increment M	M+1→M	NZ

Address mode	Op-code	Bytes	Cycles
Zero-page	$E6	2	5
Zero-page,X	$F6	2	6
Absolute	$EE	3	6
Absolute,X	$FE	3	7

INX	Increment X	X+1→X	NZ

Address mode	Op-code	Bytes	Cycles
Implied	$E8	1	2

INY	Increment Y	Y+1→Y	NZ

Address mode	Op-code	Bytes	Cycles
Implied	$C8	1	2

JMP	Jump	Flags unaltered

Address mode	Op-code	Bytes	Cycles
Absolute	$4C	3	3
Indirect	$6C	3	5

JSR	Jump to SR	Flags unaltered

Address mode	Op-code	Bytes	Cycles
Absolute	$20	3	6

LDA	Load A	M→A	NZ

Address mode	Op-code	Bytes	Cycles
Immediate	$A9	2	2
Zero-page	$A5	2	3
Zero-page,X	$B5	2	4
Absolute	$AD	3	4
Absolute,X	$BD	3	4 or 5
Absolute,Y	$B9	3	4 or 5
(Indirect,X)	$A1	2	6
(Indirect),Y	$B1	2	5 or 6

LDX	Load X	M→X	NZ

Address mode	Op-code	Bytes	Cycles
Immediate	$A2	2	2
Zero-page	$A6	2	3
Zero-page,Y	$B6	2	4
Absolute	$AE	3	4
Absolute,Y	$BE	3	4 or 5

LDY	Load Y	M→Y	NZ

Address mode	Op-code	Bytes	Cycles
Immediate	$A0	2	2
Zero-page	$A4	2	3
Zero-page,X	$B4	2	4
Absolute	$AC	3	4
Absolute,X	$BC	3	4 or 5

LSR	Logical SR	0→(7...0)→C	N=0,ZC

Address mode	Op-code	Bytes	Cycles
Accumulator	$4A	1	2
Zero-page	$46	2	5
Zero-page,X	$56	2	6
Absolute	$4E	3	6
Absolute,X	$5E	3	7

NOP	No operation	Flags unaltered

Address mode	Op-code	Bytes	Cycles
Implied	$EA	1	2

ORA	Inclusive OR	A or M→A	NZ

Address mode	Op-code	Bytes	Cycles
Immediate	$09	2	2
Zero-page	$05	2	3
Zero-page,X	$15	2	4
Absolute	$0D	3	4
Absolute,X	$1D	3	4 or 5
Absolute,Y	$19	3	4 or 5
(Indirect,X)	$01	2	6
(Indirect),Y	$11	2	5

PHA	Push A	Flags unaltered

Address mode	Op-code	Bytes	Cycles
Implied	$48	1	3

PHP	Push status	Flags unaltered

Address mode	Op-code	Bytes	Cycles
Implied	$08	1	3

PLA	Pull A	NZ

Address mode	Op-code	Bytes	Cycles
Immediate	$68	1	4

PLP	Pull status	Flags as status

Address mode	Op-code	Bytes	Cycles
Implied	$28	1	4

ROL	Rotate L	←(7...0)←C←	NZC

Address mode	Op-code	Bytes	Cycles
Accumulator	$2A	1	2
Zero-page	$26	2	5
Zero-page,X	$36	2	6
Absolute	$2E	3	6
Absolute,X	$3E	3	7

ROR	Rotate R	→C→(7...0)→	NZC

Address mode	Op-code	Bytes	Cycles
Accumulator	$6A	1	2
Zero-page	$66	2	5
Zero-page,X	$76	2	6
Absolute	$6E	3	6
Absolute,X	$7E	3	7

RTI	Return from I	Flags as pulled

Address mode	Op-code	Bytes	Cycles
Implied	$40	1	6

RTS	Return from SR	Flags unaltered

Address mode	Op-code	Bytes	Cycles
Implied	$60	1	6

SBC	Subtract	A−M−C→A	NZCV

Address mode	Op-code	Bytes	Cycles
Immediate	$E9	2	2
Zero-page	$E5	2	3
Zero-page,X	$F5	2	4
Absolute	$ED	3	4
Absolute,X	$FD	3	4 or 5
Absolute,Y	$F9	3	4 or 5
(Indirect,X)	$E1	2	6
(Indirect),Y	$F1	2	5 or 6

SEC	Set carry	C=1

Address mode	Op-code	Bytes	Cycles
Implied	$38	1	2

SED	Set decimal	D=1

Address mode	Op-code	Bytes	Cycles
Implied	$F8	1	2

SEI	Set I mask	I=1

Address mode	Op-code	Bytes	Cycles
Implied	$78	1	2

STA	Store A	A→M	Flags unaltered

Address mode	Op-code	Bytes	Cycles
Zero-page	$85	2	3
Zero-page,X	$95	2	4
Absolute	$8D	3	4
Absolute,X	$9D	3	5
Absolute,Y	$99	3	5
(Indirect,X)	$81	2	6
(Indirect),Y	$91	2	6

STX	Store X	X→M	Flags unaltered

Address mode	Op-code	Bytes	Cycles
Zero-page	$86	2	3
Zero-page,Y	$96	2	4
Absolute	$8E	3	4

STY	Store Y	Y→M	Flags unaltered

Address mode	Op-code	Bytes	Cycles
Zero-page	$84	2	3
Zero-page,X	$94	2	4
Absolute	$8C	3	4

TAX	Transfer	A→X	NZ

Address mode	Op-code	Bytes	Cycles
Implied	$AA	1	2

TAY	Transfer	A→Y	NZ

Address mode	Op-code	Bytes	Cycles
Implied	$A8	1	2

TYA	Transfer	Y→A	NZ

Address mode	Op-code	Bytes	Cycles
Implied	$98	1	2

TSX	Transfer	SP→X	NZ

Address mode	Op-code	Bytes	Cycles
Implied	$BA	1	2

TXA	Transfer	X→A	NZ

Address mode	Op-code	Bytes	Cycles
Implied	$8A	1	2

TXS	Transfer	X→SP	Flags unaltered
Address mode	Op-code	Bytes	Cycles
Implied	$9A	1	2

Appendix C2
6502/6510A Instruction Set: Classification by processor flag

Updates N, Z and C flags:
ADC,ASL,CMP,CPX,CPY,ROL,ROR,SBC.

Updates N and Z flags:
AND,DEC,DEX,DEY,EOR,INC,INX,INY,LDA,
LDX,LDY,ORA,PLA,TAX,TAY,TYA,TSX,TXA.

Updates N, Z, C and V flags:
ADC,SBC.

Updates N, C and clears N:
LSR.

Op-codes not mentioned above either: (a) have no effect on processor flags
or
(b) set or reset certain flags by direct programming
(CLC,CLD,CLI,CLV,SEC,SED,SEI).

Appendix C3
6502/6510A Instruction Set: Classification by addressing modes

---

Immediate:
ADC,AND,CMP,CPX,CPY,EOR,LDA,LDX,LDY,ORA,SBC

Zero-page:
ADC,AND,ASL,BIT,CMP,CPX,CPY,DEC,EOR,INC,LDA,LDX,
LDY,LSR,ORA,ROL,ROR,SBC,STA,STX,STY

Zero-page,X:
ADC,AND,ASL,CMP,DEC,EOR,INC,LDA,LDY,LSR,ORA,ROL,
ROR,SBC,STA,STY

Absolute:
ADC,AND,ASL,BIT,CMP,CPX,CPY,DEC,EOR,INC,JMP,JSR,
LDA,LDX,LDY,LSR,ORA,ROL,SBC,STA,STX,STY

Absolute, X:
ADC,AND,ASL,CMP,DEC,EOR,INC,LDA,LSR,ORA,ROL,ROR,
SBC,STA

Absolute, Y:
ADC,AND,CMP,EOR,LDA,LDX,ORA,SBC,STA

(Indirect, X):
ADC,AND,CMP,EOR,LDA,ORA,SBC,STA

(Indirect), Y:
ADC,AND,CMP,EOR,LDA,ORA,SBC,STA

Accumulator:
ASL,LSR,ROL,ROR

Implied:
BRK,CLC,CLD,CLI,CLV,DEX,DEY,INX,INY,NOP,PHA,PHP,PLA,
PLP,RTI,RTS,SEC,SED,SEI,TAX,TAY,TSX,TXA,TXS,TYA

Relative:
BCC,BCS,BEQ,BMI,BNE,BPL,BVC,BVS

The following instructions have no effect on status flags:
BCC,BCS,BEQ,BMI,BNE,BPL,BVC,BVS,JMP,JSR,NOP,PHA,PHP,
RTS,STA,STX,STY,TXS

Appendix C4
6502/6510A Instructions in order of common usage

In common use:
ADC BCC BCS BNE CLC CMP CPX CPY DEX DEY

INX INY LDA LDX LDY RTS
SBC SEC STA STX STY TAX TAY TYA TXA

Often used:
BEQ ASL BMI BPL DEC INC JMP JSR LSR PLA
PHA ROL ROR

Sometimes used:
AND BIT BRK BVC BVS CLV EOR NOP ORA

Seldom used:
CLD PHP PLP RTI SED SEI TSX TXS

Note: The above classification must not be taken too seriously. It is very much a question of personal preference and programming style. It is doubtful if two writers would ever agree. However, it may still be useful, particularly if you are in the initial learning phase.

Appendix D
Colour Code

Colour	Decimal	Hex
Black	0	0
White	1	1
Red	2	2
Cyan	3	3
Purple	4	4
Green	5	5
Blue	6	6
Yellow	7	7
The following colours are not valid in multicolour character mode:		
Orange	8	8
Brown	9	9
Light red	10	A
Grey 1	11	B
Grey 2	12	C
Light green	13	D
Light blue	14	E
Grey 3	15	F

Appendix E
ASCII Code

Character	Dec	Hex	Character	Dec	Hex
!	33	21	@	64	40
"	34	22	A	65	41
#	35	23	B	66	42
$	36	24	C	67	43
%	37	25	D	68	44
&	38	26	E	69	45
'	39	27	F	70	46
(	40	28	G	71	47
)	41	29	H	72	48
*	42	2A	I	73	49
+	43	2B	J	74	4A
,	44	2C	K	75	4B
-	45	2D	L	76	4C
.	46	2E	M	77	4D
/	47	2F	N	78	4E
0	48	30	O	79	4F
1	49	31	P	80	50
2	50	32	Q	81	51
3	51	33	R	82	52
4	52	34	S	83	53
5	53	35	T	84	54
6	54	36	U	85	55
7	55	37	V	86	56
8	56	38	W	87	57
9	57	39	X	88	58
:	58	3A	Y	89	59
;	59	3B	Z	90	5A
<	60	3C	[	91	5B
=	61	3D	\	92	5C
>	62	3E	]	93	5D
?	63	3F	^	94	5E

Character	Dec	Hex	Character	Dec	Hex
	95	5F	o	111	6F
£	96	60	p	112	70
a	97	61	q	113	71
b	98	62	r	114	72
c	99	63	s	115	73
d	100	64	t	116	74
e	101	65	u	117	75
f	102	66	v	118	76
g	103	67	w	119	77
h	104	68	x	120	78
i	105	69	y	121	79
j	106	6A	z	122	7A
k	107	6B	{	123	7B
l	108	6C	|	124	7C
m	109	6D	}	125	7D
n	110	6E	~	126	7E

Appendix F

How Simple Variables Are Stored

Appendix G
Glossary of Terms

absolute address: the numerical number identifying an address.
accumulator: the main register within the microprocessor and the only one equipped for arithmetic.
ACR: abbreviation for Auxiliary Control Register. One of the VIA registers.
active high: any input which requires a logic 1 to turn it on.
active low: any input which requires a logic 0 to turn it on.
address bus: the 16 lines from the microprocessor which activate the selected memory location or device.
address: a number which is associated with a particular memory location. This number can be in decimal or hexadecimal.
and gate: a gate which delivers a logic 1 out only if all inputs are logic 1.
anding: using a mask to ensure selected bits become or remain 0.
assembler mnemonics: a three-letter group uniquely defining an op-code.
assembler: a program which converts a program written in assembly code to the equivalent machine code.
base address: the operand address of an indexed instruction.
base: the number of different characters used in a counting system. Decimal is base 10, binary is base 2 and hex is base 16.
bit: one of the two possible states of a binary counting system, 1 or 0.
block diagram: a simplified diagram of an electrical system using interconnected labelled boxes.
Boolean algebra: an algebraic notation, introduced by George Boole, for manipulating two-state logic.
bubble sort: sorting an array by pairs at a time until all data is in order.
bus: a collection of wires having some common purpose such as data bus, address bus and control bus.
byte: a group of 8 bits.
Centronics: trademark for a standardised parallel interface for printers.
chip: accepted slang for an integrated circuit.
compiler: system software which translates a program written in high level language into a machine code equivalent. The entire program is translated before it is run.

conditional assembly: when parts or all of the assembled code can vary depending on test conditions.

darlington: a two-transistor configuration used to multiply the current gain.

data bus: the 8 lines from the microprocessor which carry the data to and from memory or external devices.

DDRB: abbreviation for Data Direction Register B. One of the VIA registers.

decimal: the normal counting system using the ten characters 0,1, ...9.

decoder: a logic device with many possible outputs, only one of which can be activated at a time. This depends on the logic pattern applied to the 'select' inputs.

direct addressing: the operand is a two-byte address as distinct from zero-page addressing which is a single byte address. Also called absolute addressing.

disassembler: a program which will display a machine code program in assembly language. The opposite process to assembly.

effective address: the sum of the base and relative address.

exclusive or gate: a gate which delivers a logic 1 only if the inputs are at different logic states.

exclusive oring: using a mask to ensure that selected bits assume the opposite state.

*firmware:*programs already in ROM.

flag: a single bit used to indicate whether something has happened or not (see program status register).

handshaking: a term used to describe the method of synchronising an external device to the computer.

hardware: all the bits and pieces of a computer such as the chips, circuit board, keys, etc. That which you can see, feel and break.

hex: see *hexadecimal.*

hexadecimal: a counting system using sixteen characters 0,1,...9,A,B,C, D,E,F.

high-byte: the most significant half of a two-byte number.

high level language: a language written in the form of statements, each statement being equivalent to many machine code instructions. BASIC is a high level language.

IER: abbreviation for Interrupt Enable Register. One of the VIA registers.

IFR: abbreviation for Interrupt Flag Register. One of the VIA registers.

immediate addressing: the operand is the data itself rather than an address.

implicit address: see *implied address.*

implied address: an address which is inherent in the op-code, therefore requiring no following operand.

index register: either the X or Y register when used to modify an address.

indexed address: an address which has been formed by the addition of an index register's contents.

indexed indirect addressing: the indirect address is the sum of the operand and contents of Y.

indirect addressing: the operand refers to an address in page zero which is the address of the wanted data.

indirect indexed addressing: the indirect address is modified by the addition of Y.

instruction register: a register within the microprocessor holding the op-code during instruction decoding.

integer: a whole number without a fraction.

integrated circuit: a chip containing a number of interconnected circuits.

interpreter: system software which translates and executes each high level language statement separately. BASIC is normally interpreted although compiler versions exist.

IRB: abbreviation for Input Register B. One of the VIA registers.

kernal: the 8k operating system of the Commodore 64.

latch: a buffer register which retains old data until new data is enabled.

logic gates: electrical circuits which behave as switches. The input conditions determine whether the switch is 'open' or 'closed'.

low byte: the least significant half of a two-byte number.

low level language: a series of codes rather than a language, each line resulting in one order to the microprocessor.

lsb: the least significant bit in the byte (the right-most bit).

LSI: large scale integration. Normally taken to mean in the order of tens of thousands of circuits on a single chip. The 6502 microprocessor is LSI.

machine code: strictly, this term should be used for instructions written in binary; now used loosely to include hex coding and assembly language.

macro: a routine assembled in line each time it is called.

mask: a bit pattern used in conjunction with either AND,EOR or ORA to act on selected bits within a byte.

merge sort: similar to bubble sort but faster due to progressive halving of the array before sorting into pairs.

microprocessor: the integrated circuit which is the central processor or 'brain' of the computer. The Commodore uses the 6510A species.

microprogram: a program inside the microprocessor which informs it how to carry out each machine code instruction.

mnemonics: code groups chosen so we can memorise them easily.

MOB: abbreviation for Moving OBject. Any screen object which is destined to be moved.

msb: the most significant bit in the byte (the leftmost bit).

msi: medium scale integration. Normally taken to mean up to a few hundred circuits on a single chip.

nibble: a group of 4 bits.

nybble: see *nibble*.

object code: the translated version of the source code.

one's complement: a number formed by changing the state of all bits in a register.

op-code: abbreviation for operational code. It is that part of a machine code instruction which tells the computer what kind of action is required.

operand: that part of a machine code instruction which gives the data or where to find the data.

operating system: the software already in ROM which is designed to help you use the computer.

or gate: a gate which delivers a logic out if any one or more inputs are logic 1.

ORB: abbreviation for Output Register B. One of the VIA registers.

oring: using a mask to ensure selected bits become or remain 1.

OSBYTE: keyword for Operating System Byte. Allows machine code calls to the operating system.

OSRDCH: keyword for Operating System Read Character. A subroutine for reading a character from selected input systems.

OSWORD: keyword for Operating System Word. Similar to OSBYTE but allows more parameters to be passed.

OSWRCH: keyword for Operating System Write Character. Passes character to selected output system.

page one address: any address within the range 256 to 511 decimal or 0100 to 01FF hex.

PC: see *program counter*.

PCR: abbreviation for Peripheral Control Register. One of the VIA registers.

PIA: abbreviation for the 6820 Peripheral Interface Adaptor.

pixel: a small picture element.

program counter: the only 16-bit register in the 6502 (and 6510A). Contains the address of the next instruction byte.

program status register: a register containing flag bits which indicate if overflow, carries, etc. have been caused by the previous instruction.

PSR: see *program status register*.

read: to examine the existing data in a register or memory location, usually by means of LDA,LDX or LDY.

relative address: the contents of the index register.

resident assembler: an assembler which is already in ROM when you purchase the machine.

resident subroutines: those in ROM which you can use, providing you know their starting address.

ROM: abbreviation for Read Only Memory. Information stored is permanent even when the power supply is off.

rotate: similar to shift but any bit pushed out from the carry is reinserted at the other end.

rpn: abbreviation for 'reverse Polish notation', which is concerned with the order in which numeric variables are processed by a machine.

RS423: a standardised interface which passes data serially along a single line.

scrolling: movement of the screen vertically or horizontally in order to bring fresh data into view.

shift: to move the bit pattern, one place to the left or right.

signed binary: the binary system which uses the msb as a sign bit.

silicon chip: most chips are fabricated from a silicon base although some of the super-fast modern varieties may be using a mixture of gallium and arsenic.

software: general term for all programs.

source code: the program in its high level form.

sprite: a screen object destined to be moved, together with accompanying coordinate data. Similar to MOB.

SR: abbreviation for Shift Register.

ssi: small scale integration. Normally taken to mean a few circuits, often simple logic gates, on a single chip.

subroutine: a program segment which will normally have general-purpose use and which can be used in other programs.

supply rail: a wire, feeding several components with a specific voltage.

symbolic address: an arbitrarily chosen name used in place of the numerical address. It is only recognised if it has been previously assigned to this number.

tristate: logic devices which can be either in the HIGH, LOW or open circuit state. When in the open circuit state, the output of the device is transparent to a common bus line.

TTL: abbreviation for Transistor Transistor Logic, a family of compatible logic chips operating on 5 volts. First launched by Texas Instruments but soon second-sourced by other manufacturers.

two-pass assembly: passing the source code twice through the assembler. Essential if branches are to forward addresses.

two's complement: a number formed by adding 1 to the one's complement. Used for negative number representation.

unsigned integer: a binary number without using the msb as a sign bit.

user port: one of the output sockets which can be used to control your own special devices.

user subroutines: subroutines which you can make for yourself.

vector: a word in memory containing the address of an operating system routine.

VIA: abbreviation for the 6522 Versatile Interface Adaptor chip.

volatile memory: one which loses all data when power is interrupted.

write: to place new data into a register or memory location, usually by means of STA,STX or STY. The old data is overwritten by the new.

X register: a general-purpose register which can be used in indexed addressing.

Y register: similar to X register.

zero-page address: any address within the range 0 to 255 decimal or 00 to FF hex.

Answers to Self-Test Questions

Chapter One

1.1 Source
1.2(a) Source code becomes redundant after compiling. **(b)** Execution time is faster.
1.3(a) Compile time is long. **(b)** Trial and error programming is lengthy and difficult.
1.4 ALGOL and PASCAL.
1.5 The 6510A has an on-board I/O port.
1.6 The kernal.
1.7 12K.
1.8(a) 64; **(b)** 8×8.
1.9 2023,1024.
1.10 65536.
1.11 Because there are only 8 address wires on the RAM chip.
1.12 38K.
1.13 0001.
1.14 Set HIRAM to the low state.
1.15 P3,P4 and P5 on the 6510A microprocessor.

Chapter 2

2.1 Dynamic.
2.2 Faster but more costly.
2.3 Program counter.
2.4 Arithmetic and logic unit.
2.5 ADC #1 or TAX,INC,TXA.
2.6 −128.
2.7 $7F.
2.8 Unsigned binary arithmetic.
2.9 0010 0001.
2.10 False.

2.11 The C bit.
2.12 Last in first out.
2.13 Page one.
2.14 Non-maskable interrupt.
2.15 Devices have tristate outputs.

Chapter 3

3.1 Set.
3.2 Zero.
3.3 EOR.
3.4 ORA #$04.
3.5 STA #20.
3.6 Branch type.
3.7 $2012.
3.8 JMP.
3.9 This is indirect indexed form so only Y can index.
3.10 Operand must be a page-zero address.
3.11 95 23.

Chapter 4

4.1 255.
4.2 41,50.
4.3 173,00,01.
4.4 60 is RTS.
4.5 Ability to use mnemonic groups for op-codes.
4.6 *=$C234.
4.7 It is an instruction to the assembler rather than to the microprocessor.
4.8 Pseudo-op.
4.9 TXT.

Chapter 5

No formal answers.

Chapter 6

No formal answers.

Chapter 7

7.1 64,000.
7.2 One.
7.3 Set bit 5 of the VIC-II control register.
7.4 Because the colour code for black must be entered in all screen locations.

Chapter 8

8.1 Require much lower operating currents.
8.2(a) A+B=C; **(b)** A.B.C.D.(E+F)
8.3 They are faster and take less current.
8.4 True.
8.5 True.
8.6 0.
8.7 AND.
8.8 AND.
8.9 Wired-OR.
8.10 To prevent induced voltages destroying the driver transistor.

Index